A Framework for Specifying Business Rules Based on Logic with a Syntax Close to Natural Language

Christian Wolfgang Roettenbacher, né Bacherler

A thesis submitted in partial fulfilment of

the requirements for the degree of

Doctor of Philosophy

at

De Montfort University

May 2017

I CHRISTIAN ROETTENBACHER NÉ BACHERLER, declare that this thesis titled *A Framework for Specifying Business Rules Based on Logic with a Syntax Close to Natural Language* and the work presented in it are my own and original. It is submitted for the degree of Doctor of Philosophy at De Montfort University. The work was undertaken between January 2010 and May 2017.

Abstract

The systematic interaction of software developers with the business domain experts that are usually no software developers is crucial to software system maintenance and creation and has surfaced as the big challenge of modern software engineering. Existing frameworks promoting the typical programming languages with artificial syntax are suitable to be processed by computers but do not cater to domain experts, who are used to documents written in natural language as a means of interaction.Other frameworks that claim to be fully automated, such as those using natural language processing, are too imprecise to handle the typical requirements documents written in heterogeneous natural language flavours. In this thesis, a framework is proposed that can support the specification of business rules that is, on the one hand, understandable for non-programmers and on the other hand semantically founded, which enables computer processability. This is achieved by the novel language Adaptive Business Process and Rule Integration Language (APRIL). Specifications in APRIL can be written in a style close to natural language and are thus suitable for humans, which was empirically evaluated with a representative group of test persons. A useful and uncommon feature of APRIL is the ability to define reusable abstract mixfix operators as sentence patterns, that can mimic natural language. The semantic underpinning of the mixfix operators is achieved by customizable atomic formulas, allowing to tailor APRIL to specific domains. Atomic formulas are underpinned by a denotational semantics, which is based on Tempura (executable subset of Interval Temporal Logic (ITL)) to describe behaviour and the Object Constraint Language (OCL) to describe invariants and pre- and post-conditions. APRIL statements can be used as the basis for automatically generating test code for software systems. An additional aspect of enhancing the quality of specification documents comes with a novel formal method technique (ISEPI) applicable to behavioural business rules semantically based on Propositional Interval Temporal

Logic (PITL) and complying with the newly discovered 2-to-1 property. This work discovers how the ISE subset of ISEPI can be used to express complex behavioural business rules in a more concise and understandable way. The evaluation of ISE is done by an example specification taken from the car industry describing system behaviour, using the tools MONA and PITL2MONA. Finally, a methodology is presented that helps to guide a continuous transformation starting from purely natural language business rule specification to the APRIL specification which can then be transformed to test code. The methodologies, language concepts, algorithms, tools and techniques devised in this work are part of the APRIL-framework.

Acknowledgements

I would like to thank my first supervisors Dr. Ben Moszkowski for inspiring and motivating discussions and Dr. Francois Siewe for his constructive criticism and guidance during the finalization of this thesis. Further, my second supervisor Prof. Dr. Christian Facchi for his great professional, organizational and personal support and the many hours of constructive discussions in the early phases of my work and the numerous reviews undertaken. In this context, I want to express my gratitude to Prof. Dr. Hans-Michael Windisch who I recognized as a great mentor in the first years of the preparation of this work. I also want to express special thanks to my former colleagues at THI-Ingolstadt Andreas Huebner and Dr.-Ing. Martin Bornschlegl, for bringing colour into the everyday work-life. The good organizational support of the administration teams of THI-Ingolstadt and DMU shall be emphasised at this point, many thanks!

And of course my heartiest thanks go to my family and especially to my wife for her encouragement and support during all these years. This work would not have been possible without her.

To Maren, Hannah, Jakob and Baby

Publications

1. **Working Paper (2010):** Christian Bacherler, Christian Facchi, Hans-Michael Windisch: "Enhancing Domain Modelling with Easy to Understand Business Rules" Working Paper No. 19, University of Applied Sciences Ingolstadt, November 2010. ISSN: 1612-6483.

2. **Conference Talk (2011):** Christian Bacherler "Geschäftsregeln einfach und verständlich formuliert" Talk at scientific track at REConf2011, Munich, March 2011

3. **Conference Paper (2012):** Christian Bacherler, Ben Moszkowski, Christian Facchi, Andreas Huebner: "Automated Test Code Generation Based on Formalized Natural Language Business Rules", in International Conference on Software Engineering Advances (ICSEA, 2012) pp. 165-171

4. **Journal Paper (2013):** Christian Bacherler, Ben Moszkowski, Christian Facchi: "Supporting Test Code Generation with an Easy to Understand Business Rule Language", in International Journal On Advances in Software (2013) pp. 69-79

5. **(pending) Journal Paper (2017/2018):** Christian Röttenbacher, Ben Moszkowski and Francois Siewe: "Abstraction of Behavioural Business Rules using 2-to-1 Property of Interval Temporal Logic Formulae", Submitted to Springer Requirements Engineering, ISSN: 0947-3602 (Print) 1432-010X (Online) .

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

AI Artificial Intelligence

ANTLR ANother Tool for Language Recognition

AOP Aspect-Oriented Programming

APRIL Adaptive Business Process and Rule Integration Language

AST abstract syntax tree

BPEL Business Process Execution Language

BPMN Business Process Modelling Notation

CASE Computer-Aided Software Engineering

CCGs Combinatory Categorial Grammars

CIM Computation-Independent Model

CNL Controlled Natural Language

CTL Computational Tree Logic

DSLs Domain-Specific Languages

EBNF Extended Backus-Naur Form

EPCs Event-Driven Process Chains

FN formal notation

GPLs General Purpose Languages

ITL Interval Temporal Logic

LESSA Language Engineering System for Semantic Analysis

LRA- logical, relational, and arithmetic-

LTL Linear Temporal Logic

M2M Model-to-Model

M2T Model-to-Text

MDA Model-Driven Architecture

MDSD Model-Driven Software Engineering

MOF Meta Object Facility

NLP Natural Language Processing

OCL Object Constraint Language

OMG Object Management Group

ORM Object-Role Modelling

PIM Platform-Independent Model

PITL Propositional Interval Temporal Logic

PSM Platform-Specific Model

QPTL Quantified Propositional Temporal Logic

QVT Query View Transformation

RE Requirements Engineering

SBVR Semantics of Business Vocabulary and Business Rules

SE SBVR-Structured-English

SoC Separation of Concerns

SQL Structured Query Language

SVMs Support Vector Machines

UML Unified Modelling Language

WS1S Weak Monadic Second-Order Theory of One Successor

XMI XML Metadata Interchange

XML Extensible Markup Language

Chapter 1

Introduction

1.1 Motivation and problem statement

Requirements Engineering (RE) has evolved to become the vital, initial discipline that accompanies a software system throughout its entire lifecycle. RE is utilized to answer the questions on "what" should be implemented in a software system that best satisfies the needs of a customer rather than "how" the implementation should look like. For a software project to be successful, it is crucial that the specification documents created during the RE describe the customer's wishes as accurately as possible. Typically, software projects encounter severe market competition and must be optimized among areas of tension such as demands for shortened time to market, growing development costs, shortening amortisation cycles, growing complexity and interconnectivity, and so on. Due to these forces, the importance of well-elaborated and precise requirements specifications becomes obvious. Requirements specifications should provide for conveying domain knowledge from the customer to the developers, who are then able to transform the specifications into software code. Hence, RE documents are a medium for bridging the gap between the domain experts describing a problem space and the developers creating the solution for them. Regardless of the development process used for turning a requirement into an implemented software solution, the basic process is that the transition from specification to solution, in spite of any errors or ambiguities in the specification, has to be brought about out by hook or by crook. It is generally acknowledged that the later an error is found, the more expensive it is to fix it. In the high-pressure environments affecting software projects nowadays, errors in requirements engineering increasingly pose threats to the overall success of the project.

Considerable research has been conducted in the last decades to overcome this. Much of it addresses the flaws that requirements specifications suffer from. A widely held consensus is that if natural language is used, the degree of ambiguity and imprecision often increases to an unhealthy extent. In consequence, the languages and frameworks used to address the semi-formal or formal approaches suitable for expressing requirements with precise semantics are often too complex to be used by domain experts.

Formulating a problem of a respective domain requires one to think in terms of the semantic constructs provided by the formalism. The more complicated the formalism, the more expensive is the trade off for domain experts working in an industrial project environment which enforces strict time limitations. In those cases for which a tooling for formalism exists, it may also be necessary to comprehend its mechanisms, thus further raising the entry hurdle. This is especially the case with frameworks that maintain an inherently complex meta-model which is mostly used as a semantic abstraction layer for the underlying formalism.

Of course, there are also other approaches that try to radically counteract this proposition. *Natural Language Processing (NLP)*, for example, aims at automating the transformation of natural language documents into a semantic formalism. In the best cases, however, NLP methods produce such transformations only with a probability of correctness below 1, which is in most cases unacceptable to the expectations of software architects. Each of the two trends favours one faction. On the one hand, there is the purely formal world which favours skilled developers or mathematicians, and on the other hand, there is the fully automated NLP approach which tries to remove any formalization effort, thus qualifying it to be utilized by business people.

Fortunately, there are also semi-formal approaches which try to bring the two worlds together. This can be achieved by adopting the use of suitable languages for certain modelling tasks. A prominent example for this is the *Unified Modelling Language (UML)* which comprises several graphical and textual languages. By using the UML-class model, which specifies domain specific concepts and their associations, in combination with a constraint language (*OCL*), which defines certain restrictions for the possible populations of domain concepts, it is possible to formulate business rules that are more understandable by business people. The rationale of OCL is based on predicate logic, and its formal syntax allows building very powerful and complex statements. It has surfaced that such textual constraints within OCL's artificial syntax favour the creation of complex, monolithic statements, which hence rapidly diminishes the understandability, even for

trained experts.

Therefore, the question this work attempts to answer is:

"How to close the gap in understandability between the natural language and logical formalisms both used in requirements engineering? Furthermore, how can the quality of requirements specifications be enhanced?"

1.2 Research objectives

The aim of this work is to elaborate on a framework that comprises a language called *APRIL* as well as an appropriate tool chain that supports also formal techniques for consistency checking and an abstraction of behavioural rules. A methodology on how to use the different components and formalisms is to expose the framework's applicability in a practical context and reveal the skills needed to undertake the proposed steps. Achieving this aim requires fulfilling the following objectives:

- Elaborate the core aspects for a language that is formal enough to be processed by computers and, at the same time, as close to the natural language that business people can understand.
- Define a formal semantics for the language constructs.
- Demonstrate that the new language is ahead in expressiveness, also in commonly used business rule statements.
- Define a methodology to create formal business rules from natural language specifications.
- The proposition of a toolchain suitable to support the utilization of the framework.
- Outline techniques that increase the quality of requirements specifications using consistency checking.

The novelty of the presented framework is the language and its incorporation into the requirements engineering process along with the use of a novel formalism called ISE (see Section 7.5) based on Interval Temporal Logic for abstracting formalized business rule statements.

1.3 Research methodology

APRIL features an intuitive sentence pattern technique that allows intuitively creating sentences with placeholders that can be used like functions as known from programming languages. For the semantic underpinning of the sentence patterns, atomic formulas are used, which have a denotational semantics based on OCL and Tempura.

The challenges of this thesis are:

- to investigate whether the language is ahead in understandability when compared to OCL,
- demonstrating how the framework around the language and the formalisms used can be profitably applied in requirements engineering,
- revealing which skills are needed for a user to be able to apply the developed techniques.

In general, the research can be divided into the following steps:

- *Background review*: A critical investigation of and discussion on the (semi-)formal techniques used in the state-of-the-art requirements engineering. This is followed by a validation step, reflecting how these formalisms are distributed practically in paradigms, formalisms, frameworks and tools. The aim is to identify the critical success factors of a framework suitable for contributing to requirements engineering.
- *Developing the techniques*: To support the aforementioned aim, relevant techniques are to be developed for materializing the means catering to human understanding

and interpretation by machines. This requires an appropriate formalism in the requirements specifications that is able to maintain the expressive power needed for business rules. In addition, novel approaches to raising the quality of the formal specifications are explored and discussed.

- *Identifying relevant ways to apply these techniques in practice*: The next step to bring the developed techniques into practice starts with investigating which means may be suitable for human users and processable by machines. This means that a formalism describing natural language has to be found that allows creating well-formed, unambiguous statements. In the best case, the formalism can be applied to established computer processing techniques for transforming syntax concepts into semantic targets. Moreover, developing the means to tailor natural language statements to certain domains is necessary for ensuring understandability.

- *Validation of the techniques*: Evidence for the claim that the new language is likely to be comprehensible, compared with its target language, is provided by an acceptance study with people with a basic understanding of logic. The core techniques developed are shown as technically feasible by the demonstrators. Finally, a case study containing the formalisms used shows that the overall framework is applicable.

1.4 Success criteria

The criteria for measuring the success of the work are as follows:

- Elaborating the criteria for a formal language to be suitable for requirements engineering

- Defining the syntax and semantics of this language

- Defining how a framework can be used for implementation and practically contribute to a software development process

- The capability of the framework to contribute additional value for requirements engineering

1.5 Thesis outline

This thesis is organized as follows:

- Chapter 1 introduces the context and the motivation of this thesis and gives a short overview of the research objectives and methodology, success criteria, and structure of this thesis.

- Chapter 2 presents an introduction to the state of the art in requirements engineering and describes the importance of testing in software engineering and its correlations to requirements engineering. Moreover, it gives a general impression of the abilities of a requirements engineer and outlines the challenges and quality criteria of a modern requirements engineering. The aim of the chapter is to introduce the rationale of the techniques, formalisms and tools utilized, leading up to the concepts elaborated in this work that help meet the quality criteria expected from requirements specifications. Therefore, the Object Constraint Language and Interval Temporal Logic are introduced to formalize a business that is based on predicate and temporal logic. Finally, Chapter 2 discusses related work.

- Chapter 3 discusses the different paradigms, tools, and formalisms that have evolved to bridge the gap between requirements engineering and software engineering. The aim of this chapter is to identify the core criteria for a framework suitable for use in requirements engineering that will meet and validate the state of practice of several standards and tools.

- Chapter 4 introduces and presents the motivations for the new language. In this context, the basic mechanisms specifying business rules in APRIL are described with examples given. A starting point is to describe the universe of discourse

for the rules based on UML class models. Rules for syntax beautification are sketched out and introduced for making some of the basic statements more concise and comprehensible for human readers. Common constraints, also presented in this chapter, are also motivated by the attempt to improve the language statements. However, the common constraints are far more than syntactic sugaring as each of them represents a more complex semantic construct aiming at a better expressiveness of APRIL. In sub section 4.6 the rationale of the new language, APRIL, including the syntax definition and its denotational semantics are presented. Further, a description of the basics of the target languages is presented.

- Chapter 5 describes some of the technical aspects of preparing APRIL statements from the parsing which introduces translation into code generation. Thus, an example of a mixfix parser implementation is presented. For integrating the compiler, domain modelling editors, and testing components, this chapter also contains a draft version of the compiler architecture.

- Chapter 6 explains the steps to transform a natural language requirements specification document into the language of the developed framework. In addition, the process environment for the transformation approach is presented.

- Chapter 7 validates how a novel technique based on temporal logic can improve the quality of behavioural business rules.

- Chapter 8 presents a case study conducted with thirty students, in which the newly defined language is found to be more comprehensible than the Object Constraint Language defined in the Unified Modelling Language standard.

- Chapter 9 discusses the contribution and results of this work, along with sketching out the possible future work.

Chapter 2

State of the art in requirements engineering

"Requirements Engineering is the systematic approach to developing requirements through an iterative cooperative process of analyzing the problem, documenting the resulting observations in a variety of representation formats, and checking the accuracy of the understanding gained. ([63] p. 13)"

2.1 Introduction

Globalized markets act, thus forcing enterprises in the IT sector to raise their efficiency. This is also valid for the field of software engineering as the realized turnovers are strongly scaled by manpower. The typical software engineering process is supported by standards, methodologies, techniques, and tools that help codify business requirements as well as design guidelines, along with stakeholders' individual experiences [10]. However, the discipline which significantly caters to software engineering is a solid requirements engineering right from the start [98]. The aim of an effective requirements engineering is to cast all facets of technical user demands into a form which developers can transform into software code, mostly within a narrow schedule. Software code is a formal image of real-world business rules and business processes and may obligate a good design, good documentation, a defined performance criterion, and maintainability. In practice, however, there are indeed several problems that make it difficult to comply with all the software quality criteria. Many of them are rooted in poor requirements engineering practices [110]. Practitioners are aware that several tasks of requirements engineers bear improvement potential, which is underpinned by a market survey conducted by Mich et al. [70] in 2004, which tried to answer the question *"Which are the two things in your job you would like to do more efficiently?"* directed at requirements engineers (results in Figure 2.1). It could be shown that some of the most crucial tasks concerning "identification and modelling of user requirements" and "software testing" are also those which practitioners experience as the most problematic.

At the beginning of a (software) construction process, the identification and modelling of user requirements pose the highest danger of resulting in expensive failures in later phases. Whereas the (system) testing of the software at a late stage presents a metric for measuring the quality of the implementation against the specification. Moreover, in the continuous testing during the implementation phase, it is vital to maintain a high test coverage and to keep the specification, the implementation, and the tests synchronised, which is inherently costly.

Since the survey has been conducted in early 2000, the situation of the requirements engineering practitioners, unfortunately, did not improve significantly [30]. The reason for this may be the complexity of the task itself. Identifying the user requirements in the original user specifications is sophisticated and requires high degrees of cognitive ability and creativity due to the typically very heterogeneous syntax, semantics, and pragmatics of the specification documents. Since the machine based reconstruction of human creativity has not been achieved yet, there is also little tool support to elicit ready-to-be-implemented requirements from user original specifications from scratch. Some approaches use natural language processing (NLP) to help identify requirements (mentioned by [70] and also covered in Section 3.2.3) but they have not yet gained significant momentum.

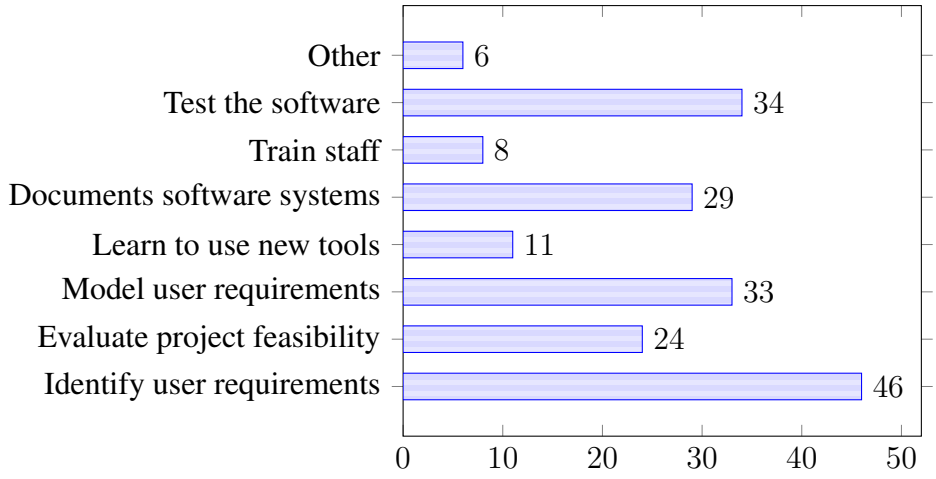


Fig. 2.1: Tasks in a requirements engineer's work with potential for improvement [70]

However, the discipline of requirements engineering (RE) has increased in importance

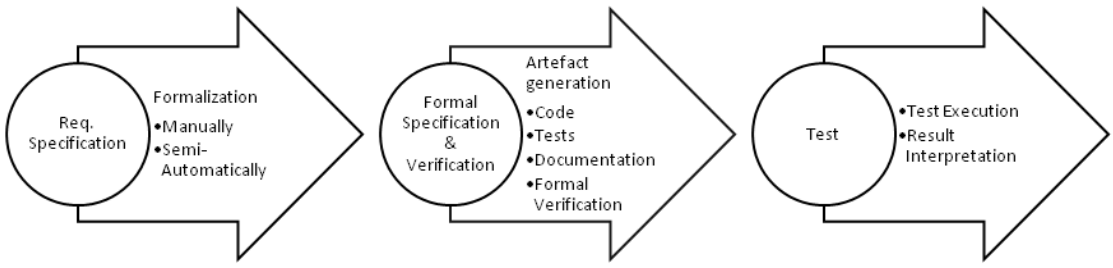


Fig. 2.2: Essential methodology in requirements engineering

over the last couple of decades [119]. Requirements engineers not only combine implicit and explicit knowledge, and communicate and negotiate with different stakeholders such as managers, administrators, domain experts, and developers [98], but are also responsible to generate digitizeable artifacts for enabling the implementation, verification, and validation of the final product (compliant with the essential methodology, see Figure 2.2).

A considerable challenge remains, which is to harmonize the communications when heterogeneous stakeholder groups collide due to differences in knowledge, mindsets, interests, resources, and languages. For instance, domain experts codify their requirements as business rules and processes in the form of abstract specifications written in imprecise, ambiguous natural language. In the best case, these natural language specifications are supported by graphical models (e.g. UML [84], Petri nets or proprietary graphs), images, or tables. Developers, in contrast, are facing the challenge to cast those specifications into software code, which requires accuracy. This is still a considerable source of failure in the interaction between the stakeholder groups. Fortunately, the discipline of modelling user requirements has a better tool coverage (some are presented in Chapter 3). Generally, a profound elaboration of the optimal tool support for individual projects binds resources if conducted with care. This imposes additional costs on a project, which is typically bound to be on a tight schedule and limited budget. It is no wonder if the choice of the optimal development process, tooling, and notation (e.g. programming language or modelling techniques) loses out on the trade-off between the elaboration of a proper tool support and the early gain of marketable results.

One reason may be that most of the time, the effort learning to handle a tool does not pay off [70]. Not using tool support for formalization increases the danger of the implementation diverging from the specification, which could be counterbalanced by automated test code generation. This becomes even more severe over time as applications evolve. But do we not have the requirements engineer in the industrial practice, to make everything good? Principally yes, but the border between raising

or reducing the efficiency of requirements engineering is blurry as it strongly depends on the individual skills of the requirements engineer [98]. Research tries to provide answers offering solid solutions to generate synergies from the mix of requirements engineers, methods, and notation standards. This is for resolving the imprecision and ambiguity of natural language specifications and raise formality. Reverse approaches are pursued as well [15] [13], trying to make formal, math-heavy notations or even programming languages appropriately readable for the originators and authors of business logic, namely businesspeople and requirements engineers. However, this is only a one-direction approach, which might be considered too half-hearted for solid requirements engineering. Enabling requirements engineering to contribute to modern development is still an active field of research and practice as well.

Considerable work has been carried out to make requirements engineering ready for, e.g., modern paradigms like Model-Driven Software Engineering (MDSD). Moreover, methodologies [103] [98], technologies [84] [86] [109] [50] [100], techniques [25] [101] [64] and tools (see also Section 3.3) have been developed to make business processes and rules more formal in order to raise the efficiency of the requirements engineering process in a way that specification artefacts are directly used for implementation. However, there are approaches that allow formal specifications which come close to a natural language. But they are either imprecise (e.g., when using natural language processing, see Section 3.2.3) or very difficult to use when it comes to formulating real-world requirements that tend to be extensive and complex [61].

A viable approach to overcoming these obstacles is seen in a business rules language based on a natural language appropriate for requirements engineers as well as domain experts and, at the same time, sufficiently formal to be processed by computers. In this context, Halpin [43] points out that: *"It is quite a challenge to design a formal language that is rich enough to capture complex rules, while still being intelligible to non-technical people"*. He identifies the following criteria for expressing business rules [40]:

- Expressiveness

- Clarity
- Simplicity and orthogonality (no unneeded concepts, minimum of inter-concept dependencies, allowing the combining of expressions if their values are legal)
- Semantic stability (minimizing the impact of changes)
- Semantic relevance. Implementation details have to be ignored.
- Validation mechanisms for the model used. The semantics of the application and the model have to correspond to each other. For that, proper mechanisms have to be provided, e.g., by theorem provers or simulation of dynamic aspects
- Abstraction mechanisms for hiding details
- Formal foundation

From that, the main criteria for a formal, general purpose, textual business rule language are inferable, which are as follows:

- Expressiveness: The ability to cover a wide range of business rules.
- Clarity: Domain experts have to be able to understand and also formulate rules in it.
- Flexibility: Ability to hide detail and allow user-defined predicates.
- Formality: Rule text must be interpretable by the computer.

The new language called APRIL (see Chapter 4), which is presented in this work, complies with these cornerstones of language design to make a significant contribution to requirements engineering. However, the formalization of business rules does not come for free only by using of APRIL. Some basic prerequisites have to be met for the input requirements documents, which are presented in Section 2.2. The understanding of a business rule in APRIL is the application of logic based on sets of typed objects.

Therefore, a form of temporal logic is used to describe behavioural rules, whereas for non-behavioural rules predicate logic is used. For describing sets we use a standardized notation suitable for describing an ontology. The technical basis for the logic frameworks and the set theoretical theories used are described in the following sections.

2.2 Stakeholders and quality criteria of requirements

"A stakeholder is a group or individual affected by the system-to-be, who may influence the way this system is shaped and has some responsibility in its acceptance" [119]. Each stakeholder has a vital interest in that system-to-be will meet certain properties. The most important property of a system is to solve functional problems imposed by the problems arising in the business domain, which means to meet the functional requirements. Next to answering the question of "what should be solved", there is also the dimension of "how a problem should be solved" which falls in the category of non-functional requirements. In this environment, the requirements engineer elicits, evaluates, negotiates, documents and validates all kinds of requirements. Van Lamsweerde describes the requirements engineering process as a spiral model that iteratively runs through all the aforementioned tasks to incrementally produce high-quality requirements. He also subsumes the quality criteria of the requirements as follows:

- *Completeness*. The set of requirements must not leave out anything that otherwise would prevent systems that meet the defined business objectives from being constructed.

- *Consistency*. All requirements are free of contradictions and conflicts (in terms of logic)

- *Adequacy*. The problem to solve must be imminent and not too far in the future. It must be explicitly addressed by one or more stakeholders. Avoid unrealistic assumptions about the environment.

- *Correctness*. Not only in terms of conceptual correctness but also regarding assumptions of the laws in the problem world.

- *Unambiguity*. For any concept, a term should have a unique meaning and be subject to consistent usage. A requirement must not allow different interpretations.

- *Measurability*. The requirements must be precise enough to prevail against software written to test them.
- *Pertinence*. The implementation of the requirement must generate customer value.
- *Feasibility*. Requirements must be technically realizable and monetarily profitable.
- *Comprehensibility*. Any requirement must be understood by the stakeholders to whom it is addressed.
- *Structuring*. A requirements document must be structured in a way that allows an easy orientation and supports the comprehension of the business domain. Hence, it must not impose comprehensibility problems related to the structure or usage of terms.
- *Modifiability*. A local modification in the requirement should ideally manifest in a local change in the software.
- *Traceability*. The context in which a requirement was elicited, written down, possibly changed, implemented, and validated should be easy to review. The impact of any removal or adoption should be as apparent as possible. Any dependencies to other requirements should also be clear.

For meeting these quality criteria, this work focuses on the technical descriptive aspect of business rules. To do so, the investigation of what should be modelled is presented in the next sections, beginning with Section 2.3. Subsequently, the rationale of how to model this, as well as the choice of the Unified Modeling Language, is presented by comparing some state-of-the-art notations, paradigms, and tools in Chapter 3.

2.3 Modelling the universe of discourse

The universe of discourse is a domain-specific set of domain concepts and terms to describe the problems in that domain. Each domain concept is a part of the business vocabulary that is supposed to unambiguously describe the business objects. The terminology of concepts can be used to describe properties, relations between concepts, and behaviour. The following example of a business constraint uses the verb phrase term *has* to denote that two domain concepts *train* and *wagon* stand in a certain quantified relation. The quantifications are expressed by *each* and *"at most 6"*.

$$\underbrace{\underbrace{Each}_{\text{quantification}}\ \underbrace{train}_{\text{concept}}\ \underbrace{has}_{\text{verb phrase}}\ \underbrace{at\ most\ 6}_{\text{quantification}}\ \underbrace{wagons}_{\text{concept}}}_{\text{business constraint}}.$$

The intentions to collect concepts and terms in a business vocabulary and to use them in specification documents are to

- unify understanding the semantics of its content amongst all of the stakeholders,
- being able to state problems using agreed terminology, and
- avoid ambiguity in the specification documents, imposed by natural language.

A further step towards a controlled natural language is to define patterns from appropriate terms, which makes it possible to state more sophisticated problems in the business domain, which may be underpinned by (semi-) formal semantics. Simple forms of such patterns can be used to define what certain verb phrases mean. For example, in our previous business constraint, the verb "has", in the context for a train and wagons, means that a composite vehicle consists of vehicles. The pattern looks like this:

$each\ <composite\ vehicle>\ has\ <vehicle>$

A more mathematical representation of the meaning can be stated as follows:

$\forall <vehicle> \mid <vehicle> \in <composite\ vehicle>$

Here, the placeholders, e.g., $< vehicle >$, stand for a specific class of conceptual objects representing a business concept. In that particular context, the verb phrase denotes

a structural description of a part of the domain model, which is unlike the behavioural descriptions covered later in this section. Lamsweerde [119] defines conceptual objects as an instantiated entity of a domain-specific concept that is manipulated by the modelled system. Conceptual objects are the actual subjects that (semi-) formal specifications can be written for and serve as the link between the requirements engineering world, with its controlled natural language specifications, and the world of the object-oriented paradigm. Hence, conceptual objects intrinsically hold some properties typical to object orientation, which are the distinct identifiability, enumerability, the sharing of similar features (similar to objects and classes in object orientation), and they may eventually differ in their state-based values and the links to other objects. These properties allow applying logical frameworks such as predicate logic and operations on sets of enumerated conceptual objects in order to define constraints as invariants and pre- and post-conditions, e.g., whilst entities evolve during system processing. Next to associations, agents, and events, entities are sub-types of conceptual objects [119]. According to [119], the characteristics of the subtypes are as follows:

- An *entity* can be regarded as a dataset object processed in a system. Thus, it is autonomous and passive.
- An *association* links two or more other objects (entities or agents). Each object at an association end plays a defined role within the association.
- An *agent* is an autonomously acting object that manipulates other objects (unlike entities). Its behaviour can be described by state sequences.
- An *event* is a special entity that exists only in one state. Its occurrence denotes that something is happening right at that moment, which may require the immediate reaction of subscribed agents.

2.4 Using the unified modelling language for business object modelling

Modelling a real-world business object by using formalisms can cause the creation of many artefacts in that respective formalism [119] if the business concept shall be described in a way as close to the domain as possible. This is especially the case when this concept is modelled in the overall business context that it is supposed to live in. Hence, it is not always possible to objectify a business concept as one to one to its counterpart in the respective formalism. According to van Lamsweerde [119], the strict approach for formalizing business concepts would be to map the constituents of the meta-models of a business process or a network of business concepts together with the target formalism. Of course, this could be implicitly done as this mapping of the constituents of different meta-models commonly describes the translation of business concepts into objects introduced by the object orientation. For decades, some formalisms have emerged that are, on the one hand, able to describe business concepts as static data structures enriched with constraints formulated by using logic. These constraints are suitable for refining the possible object populations imposed by the static data structures that typically lack this expressiveness in the favour of conciseness and do not remain too overloaded by the means of description offered.

Such formalisms are, e.g., Halpin's Object-Role Modelling (ORM) [43] and Object Management Group's Semantics of Business Vocabulary and Business Rules (SBVR) [83]. However, the most popular and widely used notation is contained in the UML, with the UML class model in combination with the OCL, which have been conceived to support object-oriented software development. Hence, the core aspects of this work are based on UML-class models and OCL. This section describes some of the basic modelling techniques in UML and presents a mapping of conceptual objects and to UML classes. Please be aware that the term "object" in the requirements engineering world is different from that in the object-oriented world. An object in requirements engineering corresponds

to a UML-class. Whereas an object in the object-oriented world is a concrete instance of a UML-class.

2.4.1 Introduction to class models in the UML

A UML class model is used to formalize an abstract conceptual view of (a part of) a certain domain in the real world. Its notation is graphical. The core idea behind class models is to depict entities that are in relation to other entities, which seems to be a very transparent and understandable concept to humans (see also Chapter 8). According to the UML specification, a class is an abstract specification (a blueprint) for concrete objects. Classes can encapsulate properties and functionality, and each instantiated object can hold individual property values and can maintain different relations – called links – to other objects. Any computation implemented against a class model actually works on its concrete instantiated data, represented by (sets of) objects that can be linked to other (sets of) objects.

The following listing briefly presents some of the basic concepts of UML class models in terms of their notation and usage.

- A *Class* is a blueprint of the structure of an object, that describes its data fields and that contains a list of the function names, which are called methods. A description of the control flow of the methods is not given in class models. The graphical notation is depicted in Figure 2.3 by the three classes named company, employee, and department. Each class is divided into three sections, the uppermost section contains the class name, the middle section has the names of the data fields, and the bottom section is for the method signatures. Note that methods with no parameters are concluded by "()". The instantiated entity of a class is called an object as shown in Figure 2.4.

- An *Association* denotes a connection of classes. Each connection end inherently assigns a role and a cardinality to the possible instances of the adjacent class. For

example, in Figure 2.3, the association of the class employee to itself describes that an instance of this class is linked to an object of the same type, indicating that one object has the role of the *manager*, whereas the other is the subordinate employee. The cardinality "0..1" at the role end "*manager*" denotes that the respective subordinate employee object must have one manager. Please note that the "0"-cardinality considers that the object representing the chairman does not relate to a manager-employee object. It can be seen that for correctness, the model has to cover all the possible compositions occurring in that domain. Hence, for constraining such compositions, a constraint language is used (see Section 2.4.3). Here, the rather unremarkable self-association impressively shows how complex an object model can become just by referencing objects of the same type. Constraining that potential complexity by using a separate notation, besides the graphical UML-class model, is necessary to keep the graphical notation manageable and provide enough expressiveness for implementing business rules. Instantiated associations are called links (see Figure 2.4 for a notation example).

- A *Composition* is a special type of association that limits the existence of its parts to that of the whole.
- An *Aggregation* is a special type of association that denotes that a class is a part of a whole. Here no existence restrictions are obligated.

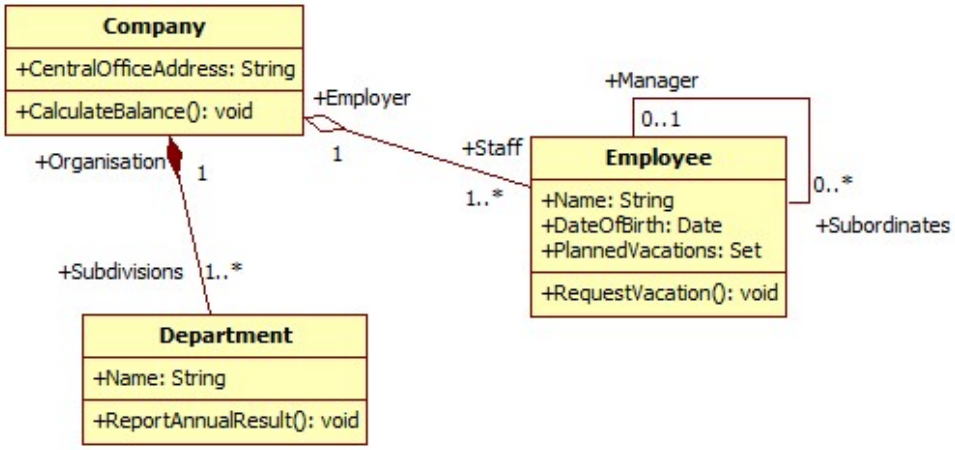


Fig. 2.3: Example UML class diagram

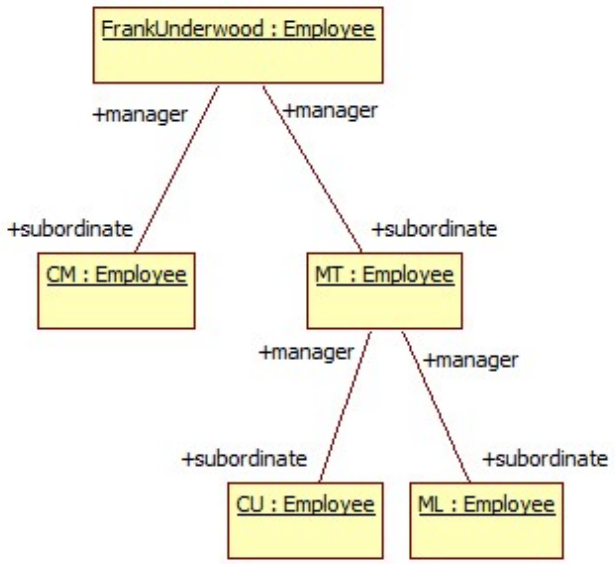


Fig. 2.4: Object model showing linked instances of class employee

2.4.2 Mapping of conceptual object types to object-oriented objects

As mentioned above, van Lamsweerde [119] strictly distinguishes between the conceptual objects introduced by a business domain and objects that are instances of classes imposed by the object-oriented paradigm (typically used in UML). The latter may be referred to as object-oriented objects. With the basic notation that UML-elements describe, it is possible to map the conceptual objects used in requirements engineering; a mapping between the two worlds is presented in Table 2.1.

Conceptual object type	**Object-oriented object**
Entity	Class, typically with no methods, which is also often referred to as a data transfer object.
Agent	Controller class with an emphasis on behaviour. Hence, a controller class contains substantial functionality of the (sub)system and holds the necessary methods, respectively. The UML compliant description would be activity diagrams or sequence diagrams.
Association	Association according to the UML specification
Event	Events and their interacting environment are modelled as a set of class patterns, depending on the type of the event. The following event types can be distinguished: asynchronous signals, method calls, state transitions, and the passing of time. Since an event requires certain stimuli to fire and causes certain effects in the system, which is reflected by the system state, events are modelled as classes in combination with the individual behavioural description.

Table 2.1: Mapping of conceptual object types used in requirements engineering to object-oriented objects

2.4.3 Brief introduction to the Object Constraint Language

The expressive power of associations and classes in UML-class models is limited to specifying conceptual models and restrictions with respect to the quantification of roles. If more details have to be specified, e.g., value ranges of certain data types or restrictions on a certain set of objects to have some pre-defined properties, OCL-constraints (Invariants, pre- and post-conditions) have to be utilized. If more detailed behavioural issues have to be dealt with, other notations are more suitable, e.g., Activity- or Sequence-Diagrams (within the UML) or, if more expressive power is needed, one has to use temporal logic frameworks (as discussed in Section 2.5).

The OCL [85, 121] is a formal, declarative language for constraining or extending UML-class models to implement business rules. It is based on first-order predicate logic and operations on sets. OCL is used to define business rules as invariants, pre- and post-conditions on a UML-class model-based object model. Pre- and post-condition rules can only be defined on methods of classes. The predicate logic statements are then defined by sets of objects or properties of individual entities, or a combination of both. Thus, for example, a filter may select certain objects that are a result of a Cartesian product, and a preceding function can then apply a logic operation which, when applied to the filtered set, will be evaluated as true or false. As with any logic based on propositional calculus, invariant pre- or post-conditions can only yield either a true or false result. However, there are implementations of OCL that support a tri-state logic [121], and the third value is denoted as undefined. Listing D.0.3 gives a small impression on how complex OCL constraints can become. OCL is part of the UML Specification [84].

Here, some example statements give an impression of the syntax. The formal syntax can be found in the relevant literature [121, 85].

The basic construction rule of an OCL constraint is as follows:

```
'context'
((ClassName 'inv') |
 (ClassName::MethodName() ( 'pre' | 'post')))
 [NameOfConstraint] :
  PropositionalStatement
```

The keyword "context" denotes the beginning of a new constraint. An invariant is introduced with a class name which is defined in the related UML-class model, followed by the keyword "inv". An alternative composition for pre- and post-conditions requires defining the class name adjacent to the method name, separated by a double colon. Optionally, for identifying the rule, the name of the constraint can subsequently be defined. The actual predicate logic business rule is defined in the rule body as a propositional statement yielding a Boolean value type. The following presents a list of natural language business rules along with their OCL representations.

- The company must have **at least one** staff member, who is the chairman.

```
context Company inv :
self.Staff->exists(member | member.Manager->isEmpty())
```

- The company must have **exactly one** staff member, who is the chairman.

```
context Company inv :
self.Staff->select(member | member.Manager->isEmpty())
->size() = 1
```

- Any employee must be an adult. The next propositional statements are based on the

UML-class model in Figure 2.3, in addition to a Class "Calendar" having a property "CurrentDate" that returns a value pointing to the current date and which is of type "Date" formatted as dd/mm/yyyy. The value accessors of "Date" are "Day" for getting the integer values behind "dd", "Month" for "mm" and "Year" for "yyyy". For using an operator on "Date", the following holds: $Date \times Date \rightarrow Date$.

```
1 context Employee inv :
2 (Calender.CurrentDate - self.DateOfBirth).Year >= 18
```

- The annual results can only be reported in September.

```
1 context Department::ReportAnnualResult() pre :
2 Calender.CurrentDate.Month = 9
```

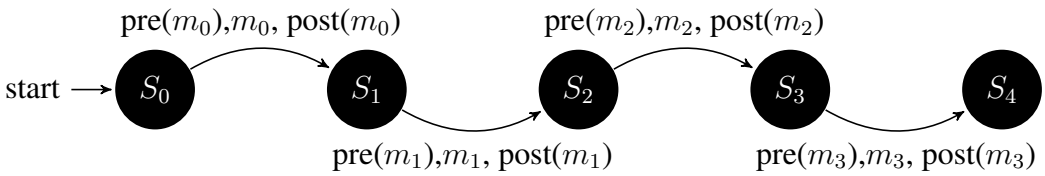
- After an Employee requests a vacation, the number of planned vacations in his schedule has to increase.

```
1 context Employee::RequestVacation() post :
2 self.PlannedVacations->size() > self.PlannedVacations@pre->
     size()
```

In OCL, the statement *@pre* denotes that the preceding statement (here *self. PlannedVacations*) points to the value before the method *RequestVacation()* was executed.

2.5 Modelling behaviour with the OCL

Behavioural descriptions are specified as sets of business concept states that change after each transition. Hence, the rationale of describing behaviour at the abstract level of a requirements specification is one of operational semantics. Certain types of behaviours can be modelled using OCL as sequences of pre- and post-conditions, each checking the system state before and after a state transition. Before the application domain for OCL pre- and post-conditions is discussed, the rationale for behavioural modelling is introduced. Say, each behaviour can be mapped to an ordered set of methods $M := m_0, \ldots, m_n$ that are defined in a UML-class model. With respect to some natural numbers $0 <= i <= n$, each method m_i triggers a state transition. A transition assigns a new value to each state variable s in a tuple of variables $S_j := s_0, \ldots, s_t$ so that $m_i(S_j)$ yields $S_{j+1} := s_0\prime, \ldots, s_t\prime$.



Assume that a sub-function P of a behaviour is defined as an adjacent pair of methods $P := \{m_j, m_{j+1}\} \subseteq M$ for some natural numbers j where $0 < j <= n$. Then this could be a case where unnecessary invalidities occur that are hard to detect in sequences of P: $post(m_j)$ and $pre(m_{j+1})$ assume different system states, such that the conjunction results in $\nvDash post(m_j) \wedge pre(m_{j+1})$.[1] For example, if more than one requirements engineer formalize different requirements, but with an intersection of certain model representations (in this case m_{j+1}), it is not obvious that a contradiction may have been introduced. Hence, contradictions in complex constraints can only be efficiently detected with an appropriate tool support, which reduces the danger of improperly designed requirements.

Limitations of OCL for behavioural descriptions:

- Contradictions in certain sequences of P, potentially invalidating the model.

[1] Here $\vDash p$ denotes a tautology, which means that p is always true. Hence, $\nvDash$ denotes the negation of it.

- Expressive Power: As Kanso at al. [51] show, it is not possible in the OCL to define temporal properties such as liveness or safety without introducing supplementary state variables into the model. However, according to Kanso et al., this "workaround" works for finite sequences but it fails with infinite sequences. Hence, liveness properties are impossible in the OCL.

- The rather complicated syntax for describing behaviour.

Because of the limitations mentioned above, we do not utilize OCL pre- and post-conditions for certain behavioural modelling applications for which we switch to Interval Temporal Logic (ITL, see Section 2.6). However, the relevant literature on OCL behavioural modelling such as [104, 51] demonstrates how OCL pre- and post-conditions can be utilized for system behaviour modelling.

2.6 Modelling behaviour with ITL

Interval Temporal Logic (ITL, see [74, 75, 76, 77]) is a linear temporal logic for reasoning on propositional and first-order statements specified upon intervals of discrete time. It was first introduced by Ben Moszkowski in the early 1980s and has since then gained momentum in several research areas dealing with formal temporal frameworks. Intervals and discrete linear state sequences offer a natural and flexible way to model computational processes involving software and hardware. ITL offers the well-known temporal operators such as "always", "sometimes", and "next" as well as additional operators such as the sequential composition "chop" and the iteration "chop-star". In general, the chop operator states that one interval is followed by another, which shares the adjacent state. This concept is analogous to the concatenation and Kleene-Star operators for regular languages and expressions. ITL can express some imperative programming constructs (e.g., while-loops) and has an executable subset called Tempura [74]. With Tempura and AnaTempura [74], there exists an executable subset of ITL that ships with a tool to animate and experiment with to-be-developed ITL statements. Moreover, there also exist

some compilers (see Gomez et al. [39]) for a subset of ITL to be used for validating the translated statements in the model checker called MONA [55], to be found on the web page [2]. Chapter 4 presents more detail on ITL and the utilization within the APRIL.

2.6.1 Rationale of using Interval Temporal Logic for describing behavioural business rules

Formal specification languages offer a range of sophisticated forms for analysing a specification document, ranging from adequacy checking of algorithmic or deductive verification of desired properties to formal completeness checks [119]. Formal notations allow the automated tool-based generation of counterexamples to formulas and also the generation of software, e.g., source code or test code [6]. Using this adds significant value to constructing a software product and provides a non-neglectable counterweight to the costs of employing formal methods in requirements- and software engineering. In this paper, we want to go further and show how reasoning on business rules that are based on Propositional Interval Temporal Logic (PITL) [77] can be applied for improving the quality of requirements specifications.

The improvement is twofold. On the one hand, there is the detection of inconsistencies (described in Section 7.4). On the other hand, we present a novel approach to deduce a simplified abstract version of a complex behavioural business rule, which contributes to a better comprehensibility of the business rule for the requirements engineer. The starting point for these techniques is an APRIL specification of behavioural business rules which are semantically based on an executable set of PITL, a subset of Interval Temporal Logic (ITL) [74] materializing in Tempura.

The reason for not considering OCL's pre- and post-conditions as a semantics for underpinning behaviour is that generally Linear Temporal Logics (and hence Tempura, too) have the advantage of not coupling the transition with the description of the behaviour, making the formulations more natural. This is not the case with OCL's pre-

[2] http://www.brics.dk/mona/publications.html

and post-conditions, which may be useful to describe the states before and after one single state. Moreover, temporal logic is very popular and very well understood to formally describe behaviours of software and hardware systems. It has a formal semantics and offers the basic temporal operators such as $\Diamond$ "*sometimes*" and $\Box$ "*always*" to denote that a proposition holds *at least once* or *always* in a linear sequence of states. Other important basic operators are $\bigcirc$ "next" to denote that a proposition holds in the subsequent state or *While* and *Until* (*W* / *U*) denoting that a proposition always holds unless/until another one is satisfied.

In practice, graph-based modelling languages, e.g., UML-state diagrams or Petri nets have become popular to model behaviour as automata, which can be checked against temporal logic formulas. However, the classical linear temporal logic frameworks sometimes lack the means to concisely express certain scenarios. This is the case, for example, when it comes to defining high-level imperative constructs such as behavioural repetitions and multiple time granularities in a software or hardware system. Therefore, Interval Temporal Logic not only introduces operators such as *chop*, *chop-star* (for repetitions) and projection but also assignments and conditionals [77]. ITL's modelling paradigm is suitable for non-temporal logic experts [39]. Thus, better comprehensible ITL specifications reduce the danger of giving a wrongly formulated system design too much confidence, as more people are able to check these specifications and potentially detect errors [39].

In the context of APRIL, the construction of a new generation of templates in Tempura is easier. Moreover, the ability to provide the full power of regular expressions to a specification language makes it more attractive [96] [120]. ITL chop and chop-star bring this expressive power to temporal logic, which is comparable to the Kleene-Star known from regular expressions. This allows defining ω-regular expressions for infinite automata, which is not completely possible with point-based LTL frameworks [120] [122]. Another important reason for generally choosing ITL is that it has gained momentum in the definition of safety and liveness properties in the last three decades.

Complex business rules formalized in conventional PTL [95] are often too complex and difficult to understand for non-mathematicians. ITL overcomes this by splitting (using *chop*) a state sequence into intervals where the first is "*followed by*" the second, and so on, and allows tailoring a minimized version of a temporal formula to a subinterval. This provides a more natural readability and a better comprehensibility of the temporal logic statements. Another important aspect for ITL (and hence Tempura) is that most business rules used to model processes can be expressed as a logical property of safety or liveness [77]. Any of such business rules is at least an element of these two kinds of properties, which is, as we present in Section 7.5, important for applying the abstraction mechanism to behavioural business rules. Moreover, PITL has the same expressive power as Quantified Propositional Temporal Logic (QPTL) for a finite time, regular expressions, and Weak Monadic Second-Order Theory of One Successor (WS1S). Especially the correspondence of PITL's and WS1S's expressive power ideally contributes to evaluating PITL formulas in the used checker tool MONA [55] [39], which is based on WS1S. One of the main reasons why we chose Tempura is because PITL's and ITL's interval properties generally allow an appealingly natural expressibility of properties, which can, in addition, be at a higher level than point-based ones. An important aspect is to call on an appropriate tool support, which is possible with Tempura and MONA for the reasoning on PITL formulas. It is particularly interesting to explore the practical applicability of certain newly discovered properties, such as the 2-to-1 property [77, 76] that are intrinsic to certain classes of PITL formulas, which we want to briefly introduce in the following subsections 7.2 and 7.3. The next section introduces ITL, PITL, and the 2-to-1 property based on PITL.

2.6.2 Syntax of ITL

The basic syntax constructs of ITL along with some of the most important formulae and expressions are presented in Table 2.2.

- int: denotes an integer value: $int ::= [0-9]\{0-9\}$

- s: denotes a static variable or global variable, respectively, that never changes over time: $s ::= [a-z]\{[a-z]|[A-Z]|\_|[0-9]\}$
- S: denotes a state variable that can change over time: $S ::= [A-Z]\{[a-z]|[A-Z]|\_|[0-9]\}$
- χ: denotes either an s or an S.
- p: is a predicate symbol such as the relational infix operators, e.g., $=$, $\geq$, $\leq$, $<$, and $>$, or other user prefix operators defined for which the following type concatenation characteristic holds: $ANY \times ANY \rightarrow BOOLEAN$
- op: denotes an operator such as +, -, *, or / to be used in infix notation or *mod* in prefix notation.
- formulae f and expressions e as in Table 2.2

$$e ::= true \mid false \mid int \mid s \mid S \mid op(e_1, \ldots, e_n) \mid \bigcirc e$$

$$f ::= p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid (\forall \chi)(f) \mid skip \mid f_1; f_2 \mid f^*$$

Table 2.2: Syntax of ITL using generic constructs

For a comprehensive presentation with detailed examples, please consult the relevant literature [74].

2.6.3 Semantics of ITL

The model of ITL is based on discrete sequences of states that can be split into intervals. An interval σ consists of individual states $s_0 s_1 \ldots s_n$, where $n = |\sigma|; n >= 1$ and $|\sigma|$ is defined as the length of the interval, which is the number of states in the interval minus one. Each state s_i maps a variable to some value of a certain type (e.g., integer or boolean). An evaluation function $\mathcal{M}_\sigma$ assigns a truth value $\{true|false\}$ to a formula *f*, taking into account the whole interval σ. Hence, $\mathcal{M}_\sigma[\![f]\!]$ is either $true$ or $false$. Intervals can be concatenated by the operator chop ";" e.g., $\sigma := \sigma_1 \ ; \ \sigma_2$ This implies the following terms:

- Prefix subinterval of σ is σ_1
- Suffix subinterval of σ is σ_2

Furthermore, the following is valid with respect to the interpretation function $\mathcal{M}_\sigma$ and the syntax definition in Table 2.2:

$\mathcal{M}_\sigma[\![S]\!] \equiv S_{s_0}$		A state variable's value S on an interval equals the state variable's value in the initial state. Thus, taking into account the derived $Next$-operator from Table 2.4, this also holds for a state variable's value S_{s_n} on an interval in the n^{th}-state: $\mathcal{M}_\sigma[\![\bigcirc^n S]\!] \equiv S_{s_n}$
$\mathcal{M}_\sigma[\![\neg f]\!] \equiv true$	iff	$\mathcal{M}_\sigma[\![f]\!]$ is $false$
$\mathcal{M}_\sigma[\![f_1 \wedge f_2]\!] \equiv true$	iff	$\mathcal{M}_\sigma[\![f_1]\!]$ is $true$ and at the same time $\mathcal{M}_\sigma[\![f_2]\!]$ is $true$
$\mathcal{M}_\sigma[\![skip]\!] \equiv true$	iff	$\|\sigma\| = 1$, which means that the interval has exactly two states.
$\mathcal{M}_\sigma[\![f_1; f_2]\!] \equiv true$	iff	there exists a j, for which $0 < j \leq \|\sigma\|$ holds, such that $\mathcal{M}_{\{s_0,\ldots,s_j\}}[\![f]\!]$ is $true$ and $\mathcal{M}_{\{s_j,\ldots,s_{\|\sigma\|}\}}[\![f]\!]$ is also $true$
	or	if σ is infinite, then $\mathcal{M}_\sigma[\![f_1]\!]$ is $true$.
$\mathcal{M}_\sigma[\![(\forall\chi)(f)]\!] \equiv true$	iff	for all sub-formulae $g_n(\chi_{s\prime}) \in f_{s\prime}$ of **state** $s\prime$, the following holds: $\bigwedge_{n=0}^{\mathbb{N}} g_n(\chi_{s\prime}) \equiv true$. Conventional universal quantification dedicated to a single **state**.
$\mathcal{M}_\sigma[\![f^*]\!] \equiv true$	iff	one of the following holds: 1. $\|\sigma\|$ is not ∞: f can be split into an infinite number of finite-length subintervals sharing adjacent states like $chop$, and each satisfies f. 2. $\|\sigma\|$ is ∞: f can be split into an infinite number of intervals sharing adjacent states, and each satisfying f.

Table 2.3: Semantics of ITL

Moreover, Moszkowski defines some derived operators, which are listed in Table 2.4.

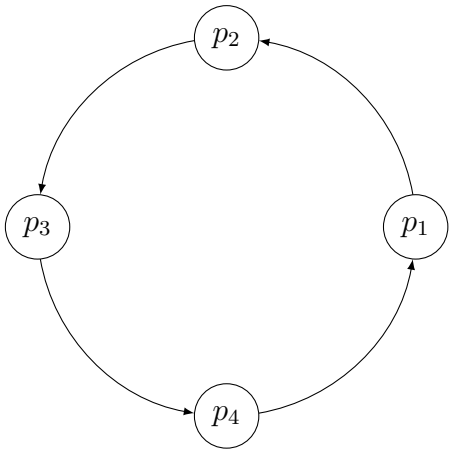
Disjunction:	$f_1 \vee f_2$	$\widehat{=}$	$\neg(\neg f_1 \wedge \neg f_2)$
Implication:	$f_1 \rightarrow f_2$	$\widehat{=}$	$\neg f_1 \vee f_2$
Equivalence:	$f_1 \equiv f_2$	$\widehat{=}$	$(f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1)$
Existential Quantifier:	$(\exists \chi)(f)$	$\widehat{=}$	$\neg((\forall \chi)(\neg f))$
Infinite interval:	inf	$\widehat{=}$	$true\ ;\ false$
Finite interval:	$finite$	$\widehat{=}$	$\neg inf$
Next:	$\bigcirc f$	$\widehat{=}$	$skip\ ;\ f$
More:	$more$	$\widehat{=}$	$\bigcirc true$
Empty:	$empty$	$\widehat{=}$	$\neg more$
Eventually:	$\Diamond f$	$\widehat{=}$	$finite\ ;\ f$
Always / Henceforth:	$\Box f$	$\widehat{=}$	$\neg(\Diamond(\neg f))$
Final state:	$finf$	$\widehat{=}$	$\Box(empty \rightarrow f)$

Table 2.4: Derived ITL Operators

2.6.4 Example of behavioural models that can be described by ITL

Unlike OCL, ITL uses discrete sequences of time points for modelling behaviour. Time points can be grouped into intervals, each of which holds for a certain ITL formula.

For example, assume that we want to describe a machine in a manufacturing plant that places an actuator one by one at a sequence of positions $P = p_1, p_2, p_3, p_4$.



The arrival of each position is reflected by a Boolean variable of the same name (p_x) switching to $true$. Additionally, at each position, it is necessary to activate a sequence of actuators. The manipulation itself requires the actuator to be triggered by a sequence of output signals $A = q_1, q_2$ and be reflected in a certain sequence of input signals $\iota =$

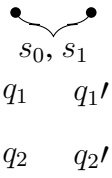
i_1, i_2, i_3 if the manipulation is conducted correctly. For the sake of simplicity, a pre-condition check preceding the execution of A is omitted.

For that, A is formally defined as an interval with two states with. As an illustration of the following formula, please consider Table 2.5 that shows the values of q_1 and q_2 in the two states s_0 and s_1. Please note q_2 is of a numerical type. The evaluation within the conjunction "$\ldots \wedge q_2 = 1006 \wedge \ \ldots$" is done on the equation operator, which yields a truth value that complies with the types expected in that composition.

$$A := skip \wedge q_1 \wedge q_2 = 1006 \wedge \bigcirc(\neg q_1) \wedge \bigcirc(q_2 = 2009)$$

	s_0	s_1
q_1	$true$	$false$
q_2	1006	2009

Table 2.5: States of A

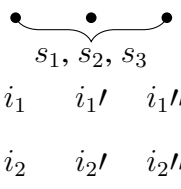


Hint: The "$\prime$" in an expression such as $q_1\prime$ denotes the value of q_1 in the next ($\bigcirc$) state.

The reaction of input signals B can be formally specified as an interval of length 2 (which means that it has three states) as:

$$B := len(2) \wedge i_1 \wedge i_2 \wedge i_3 \wedge \bigcirc(\Box\neg i_1) \wedge \bigcirc(\neg i_2) \wedge \bigcirc(\Diamond\neg i_3) \wedge \bigcirc \bigcirc (i_2) \wedge fin(i_3)$$

See Table 2.6 for a tabular representation. Please be aware that "$-$" stands for "does not matter".



	s_1	s_2	s_3
i_1	$true$	$false$	$false$
i_2	$true$	$false$	$true$
i_3	$true$	-	$true$

Table 2.6: States of B

Each sequence at a position p_x can be formalized as:

$C_x := Pos_x \ \wedge (A; B)\ ; skip$

whereas Pos_x explicitly reflects one position out of the positions {p_1, p_2,p_3, p_4}.

$Pos_1 \ = \ (p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge \neg p_4)$

$Pos_2 \ = \ (\neg p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4)$

$Pos_3 \ = \ (\neg p_1 \wedge \neg p_2 \wedge p_3 \wedge \neg p_4)$

$Pos_4 \ = \ (\neg p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge p_4)$

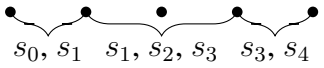
Hence, C_x represents:

$C_1 := Pos_1 \ \wedge (A; B)\ ; skip$

$\vdots$

$C_4 := Pos_4 \ \wedge (A; B)\ ; skip$

Also, any C_x holds for the following interval partitioning:



The final form of the formula describing the behaviour of the manipulator can be formalized as follows:

$(C_1 \ ; \ C_2 \ ; \ C_3 \ ; \ C_4)^*$

Starting from Position 1, reflected by Pos_1, C_1 holds for the correct input and output sequences A and B. This is followed by an interval with two states ($skip$), for which no state variables are explicitly defined, indicating that the manipulator is turning and no actuator is triggered in the second of the two intervals. In the next subinterval C_2, the same sequences A and B get replayed, which also holds for C_3 and C_4. The entire sequence C_1-C_4 is continuously passed through, on and on, again indicated by the "*" (chopstar) operator at the end.

2.6.5 Justification

Interval Temporal Logic has been used for a number verification tasks around software and hardware design dealing with behavioural descriptions. Its executable subset Tempura plays a vital role as with AnaTempura, it is possible to execute statements and check the correctness of certain to-be-developed formulae. Hence, Tempura experiences further development to keep up with the latest research on ITL. There are several reasons for choosing the ITL and in consequence, Tempura for specifying behavioural modelling in the APRIL framework. These reasons are listed below:

- Unlike OCL, Tempura is not prone to specifying potentially contradictive specifications as there is no distinction between pre- and post-condition of a method. Additionally, Tempura does not need auxiliary constructs to achieve the needed expressive power as OCL does.
- There exists an axiomatic framework to abstract (P)ITL and compliant Tempura formulae to support an abstraction of formulae. The details and Application are described in Chapter 7.
- With AnaTempura, it is possible to execute the formulae for correctness checking.
- The compiler PITL2MONA and the checker tool MONA allow verifying Tempura formulae that are compliant to Propositional Interval Temporal Logic (PITL, see Section 7.2)
- Tempura provides enough expressive power for the white box testing purposes aimed at.
- The natural reading of the formulae with the "chop"-operator (e.g., 1-to-1 verbalizable to *"followed by"* or *"and then"*).
- All the software tools used are freely available

- Profound and rigorous semantic underpinning of ITL. Moreover, ITL is a well-established and important formalism.

Please be aware that the list is not sorted (e.g., by importance) as in the context of the overall APRIL framework, the priorities may shift according to what a user wants to achieve in his/her role.

2.7 Rationale of business process modelling

Efficiently supporting business processes in Workflow Management Systems and Business Process Management systems is a success factor for companies in almost all domains. Melenowsky [66] estimates that companies that spend a significant part of their overall project time on the rigorous construction of specifications, tremendously reduce the risk of expensive system redesign in the late phases. This generally accepted insight paved the way early to a wide range of techniques supported by formal rationale as well as notations and languages, e.g., Business Process Modelling Notation (BPMN), UML-Activity diagrams, Business Process Execution Language (BPEL), Event-Driven Process Chains (EPCs) etc. [117].

In practice, a handful of notations to describe business processes have emerged over the last decades. The most popular ones pick up on the principle of the well-known EPCs. These are the BPMN and UML Activity Diagrams. Most of them can be underpinned by a semantic foundation based on Petri nets [117], which introduces beneficial properties such as verification, model checking, or process mining. Others are based on describing finite state automata such as the UML State Diagram. Whereas, the use of formal temporal logical frameworks is rather rare and is mostly used in applications designated to be formally verified against certain logical properties. In Table 2.7, a brief overview of the benefits and flaws of the basic process modelling paradigms mentioned is given.

From our viewpoint, utilizing formal method disciplines materializing in a proper tool support bears the highest potential for helping to reduce errors in process modelling tasks. In commercial software, the potentials of formal methods for process modelling are only partly implemented, and the functionalities offered are not fully utilized by the users [117]. One major issue is that using formal methods requires the system specification to be sufficiently detailed, and this tends to become incomprehensible over its entire lifecycle, which may potentially exceed the lifetime of its technological manifestation, the software system. The pressure of reducing complexity is even more intensified when several derivatives of a process are maintained in parallel [58]. In order to find a balance,

Technique	Benefit	Flaw
Concrete modelling techniques (UML sequence models, event-driven process chains EPC, Business Process Modelling Notation BPMN)	Concrete scenarios of comprehensible examples, specification by example	Covers only the concrete requirement, and is formal enough
State machines, including those using annotated transitions and/or pre- and post-conditions	A tool-based transformation to state machines enables reasoning	Can get complex very fast, handling of tooling and theory behind it have to be mastered first
Formal temporal logic frameworks with the typical mathematical notation	A tool-based transformation to state machines enables reasoning	Requires rigour in formalizing requirements, very hard to construct and maintain by non-logic experts

Table 2.7: Benefits and flaws of process modelling techniques [117]

Mendling et al. [68] describe a range of fundamental principles that help to focus on the maintainability of the business processes modelling practice. These guidelines still leave a requirements engineer or architect relying on his creativity when it comes to defining concise, comprehensible, and minimally complex process models.

With the APRIL Framework (described in Chapter 4), it should be possible to soften these flaws by reasoning that behavioural models should reduce the number of concrete elementary processes to one abstract process at a higher abstraction level. This allows better comprehensibility of the process itself and simplifies the derivation of concise acceptance criteria for testing the implementation against the specification.

2.8 Related work

This section discusses the related work dealing with the rationale and the application of formal representations for requirements.

2.8.1 Business rules and testing

As introduced, the proposed language APRIL deals with the transformation of natural language specifications close to natural language into predicate and temporal logic frameworks for testing purposes. Most of the related work discussed comprises utilizing mechanisms that aim at achieving comparable results. Generating tests from requirements specifications is an appropriate way to raise the quality of software, also shown by Demuth et al. [23]. Demuth at al. could show that, for certain kinds of requirements, the automated generation of abstract test cases is able to save about 34% of the test effort. However, they used a use case-based approach, which is at a higher abstraction level than those (including APRIL) using domain models.

A very prominent approach that aims at describing the semantics of domain model-based requirements is the SBVR specified by the Object Management Group (OMG). SBVR uses a meta model that can serve as the semantic basis of different controlled natural languages. SBVR-Structured-English (SE) and, similarly, RuleSpeak [83] are so-called controlled languages used for expressing business rules in a restricted version of a natural language. Both are based on SBVR, which defines semantic parts, e.g., the terms and facts to determine business concepts and their relations. The syntactic representation of these parts is achieved by text formatting and colouring, which could be used to aid parsing SE-statements. From our point of view, mixing technical information with the textual representation is problematic because formalized and natural language semantics have to be maintained in the same statement. However, natural language does not utilize text formatting information for transporting semantics.

Nevertheless, SE is used for model representation, which also Kleiner et al. [56]

utilize as a starting point for translating schema descriptions (in SE) into UML-class models, which is helpful for software development. Unfortunately, they leave the treatment of business rules to the further work. Regarding the customizability aspect of business statements, the approach of Sosunovas et al. [106] presents another way of utilizing regular patterns. They pursue a three-step approach to constructing business rule templates that are first defined on an abstract level and then tailored to fit a specific domain with every further refinement step. Therewith, they provide precise meta model-based semantics to the template elements but – as they admit – not to the business rule resulting from using the template.

In the field of semantic web, several Controlled Natural Language (CNL) approaches have been elaborated. Hart et al. [44] propose a CNL called Rabbit to specify an ontology. The language provides the means to specify concepts and relations in a dictionary-like manner. Axioms describe the kind of relations between concepts and also allow specifying cardinalities on relations and constraints based on propositional logic. Moreover, Rabbit allows referencing another ontology to make use of already existing concepts and axioms.

A pragmatic approach to defining natural language constructs in CNL is presented by Spreeuwenberg et al. [108] and van Grondelle et al. [118]. They use patterns with a regular syntax consisting of constants and placeholders that can be replaced by instances of meta-model concepts. Each pattern is related to a graph in the meta model to represent its semantics exclusively based on that meta model. However, from our viewpoint, the interesting thing is that they highlight the particular simplicity of the construction of patterns even for untrained persons. This is also what surfaced in the first usability tests of the APRIL definitions.

Another interesting approach for generating tests from requirements specifications is introduced by Nebut et al. [80]. They utilize UML use-case models combined with contracts represented by pre- and post-conditions to specify sequences of state transitions. Based on these contracts, they simulate the modelled behaviour by intentionally

"instantiating" the use-case model. This approach could be a worthy extension to ours, which uses historical data that could also be generated by simulation. Moreover, Nebut et al. show how to generate test cases from sequence diagrams and test objectives dealing with a defined test coverage.

A recent approach undertaken by Salemi at al. [102] seems to tie it all together. They are using a three phase approach starting from natural language processing in the initial phase to construct an abstract syntax tree compliant with a simplified version of natural English, on which they also defined the grammar rules. Therefore, they define a suitable simple subset of English syntactic structures which they apply on a given set of business rules written in a form of natural English that is structured very simple. For generating an intermediate SBVR representation in the second phase, the pre-processed business rules are mapped to the corresponding SBVR syntactic concepts using their predefined set of mapping rules. Finally in the third phase, Salemi et al. generate OCL rules based on the SBVR model instance.

The evaluation of their approach surfaces an intriguingly high accuracy of correctly transformed natural language sentences. Even if the achieved accuracy, ranging from 99% for very simple constraints with a very simple complexity (e.g. recognizing a number) down to round about 74% for higher complex constraints, this might be not sufficient in many use cases. The time saved for formalizing requirements using automatisms such as natural language processing seems very appealing especially in the context of the presented numbers by Salemi et al., who mention a time saving ratio nearly up to factor 3 in the best case. However, from our viewpoint, also here the accuracy of natural language processing leads to the insight that final human supervision is needed in this context. Moreover, Salemi at al. [102] do not show how behaviour is addressed, nor how the expressiveness of the semantic underpinning can be extended.

2.8.2 Process modelling

Approaches that try to extract abstract models from SBVR specifications using natural language processing suffer from imprecision [45]. Regarding the customizability aspect of business statements, the approach of Sosunovas et al. [106] presents another way which utilizes regular patterns. They pursue a three-step approach for constructing business rule templates that are first defined on an abstract level and then refined stepwise to fit a specific domain with every further refinement step. Therewith, they provide precise meta model-based semantics to the template elements but – as they admit – not to the business rule resulting from using the template. A pragmatic approach for defining natural language constructs in CNL is presented by Spreeuwenberg et al. [108] and van Grondelle et al. [118]. They use patterns with a regular syntax consisting of constants and placeholders that can be replaced by instances of meta-model concepts. Each pattern is related to a graph in the meta model to represent its semantics exclusively based on that meta model. However, the interesting thing is that they highlight the particular simplicity of the construction of patterns even for untrained persons. This is also what surfaced in the first usability tests of the APRIL definitions.

In the sector of consistency checking for business processes, a range of contributions addresses the validation of processes formalized in BPMN using Computational Tree Logic (CTL), a form of temporal logic. Here the representative work of Awad et al. [4], who present a method for compliance checking of business processes, is based on CTL formulas that have to be specified by a user. The CTL compliance rules represent the domain-specific guidelines which a process that is composed of control and data-flow artefacts has to follow. Unlike the approach for consistency checking elaborated in this thesis, Awad et al. allow specifying the definition of anti-patterns, which is also the reason why they use CTL, as Awad et al. mention that anti-patterns are not to be formalized in Linear Temporal Logic (LTL) and hence ITL. Another difference in the approach of this thesis is that finding inconsistencies is completely independent of any domain-related rule as the behavioural rules only get evaluated in a single-variety context. This means that

rules to restrict other rules are intentionally outside the scope of the APRIL framework. In APRIL, the evaluation is meant as the detection of contradictions (predicates that can never hold) or tautologies (predicates that are always true). However, the ability to specify anti-patterns is an interesting aspect for the consistency checking of behavioural business rules as it provides the user with the ability to specify states which are known to be explicitly prohibited. This can shrink the allowed state space for a business rule, thus significantly reducing the effort to be taken for model checking and repeatedly readapting a certain behavioural rule.

The formal underpinning of the simplification of behavioural business rules that are based on the classical point-based LTL frameworks is addressed by a range of academic formal methods literature such as Micheal Fisher's textbook [31], which introduces a clausal temporal resolution of propositional temporal formulas, supported by the TSPASS tool. The basic principle of this method is renaming, which was first described by Plaisted et al. [94]. The consequent application of clausal temporal resolution deduces a propositional temporal logic formula, first translated to a normal form, to a stage where inconsistencies are easy to detect. Other approaches using clausal deduction on linear time temporal logic formulae, such as Bolotov et al. [9], use classical elimination and introduction techniques to guide proof strategies. However, we have not found an approach that introduces simplification rules and models time in a way that is as similarly abstract as with intervals. Nevertheless, clausal deduction could help to find generalized abstracted formulae that could be used in our simplification process.

Approaches that address the verification of requirements using model checkers impose the need of transforming natural language specification into a formal language [53] such as the language called "Z". Here, the danger of divergence between intention and implementation is always present when manually transforming non-formal to formal notations. This is addressed by the APRIL framework, which guides the transformation by a methodology suitable to detect errors at an early stage.

The first step is to find reusable and approvable patterns, which is aimed at by

approaches using normalized, regulated forms of rewriting as done by Denger et al. [24] or Tjong et. al [113]. They introduce natural language patterns with annotations that are based on metamodels and logical rationale. The semi-formal underpinnings introduced help to wipe out imprecision imposed by using natural language. This is done by providing a schema for the systematic rewriting of specifications when applying the pattern frameworks onto the original requirements. However, these presentations are not formal enough to be used by computers. Interestingly, Tjong et al. present a list of useful rules for creating and applying patterns, which are pretty close to that what we experienced when we transformed natural language business rule statements into APRIL Definitions.

The second step is to generate computer-interpretable artefacts generated by using the templates. These artefacts are actually OCL and Tempura statements to be used as test code against a productive implementation. Thereby, the productive code is prepared to generate historical data at suitable positions during execution, which can be evaluated by an OCL runtime and AnaTempura, respectively taking into account the aforementioned test code statements. A similar approach is pursued by Nebut et al. [80], who as mentioned earlier. Next to several others, Xu [125] presents in his approach to generating test code based on high-level Petri nets models, which is suitable for detecting errors in code that represents behaviour.

However, the definition of a seamless transformation from pure natural language statements down to executable test code, along with the ability to extend the language's expressive power, are unique within the APRIL framework. So is the combination of the entire feature set, listed as follows:

- Mixfix notation for reusable rule definitions to mimic natural language sentence patterns
- A language that is suitable to be understood by business people
- Domain-independent rules that occur very frequently in any modelling context; the

so-called Common Constraints

- The ability to extend the grammar by adding EBNF rules along with the translation templates for the respective target language.
- A methodology to stepwise transform natural language specifications into formalized statements of business rules represented in the APRIL language
- Methods for consistency checking and abstracting behavioural business rules

2.9 Distinction to other Testing Pardigms

APRIL can be used to generate test code based on OCL and Tempura, which is discussed in the later chapters. Testing with APRIL implies that historical data gathered from a productive software system (in this case the system under test (SUT)) is used to check whether the SUT behaves as specified in the APRIL business rules. From the perspective of this work, that focuses on the APRIL language along with methods using it, working with historical data has several advantages. One mayor advantage is that the runtime performance does not play a big role for the implementation of the test generation and execution system.

However, since APRIL can be used for test code generation, there has to be a disquisition of other paradigms dealing with test code generation. Therefore, the following comparison considers model based testing (MBT) originating from Chow's W-Method [19] and category partitioning presented by Ostrand et al. [90]

2.9.1 Model Based Testing

Instead of discussing each aspect of model based testing in detail, it appears reasonable to outline some similiarities between APRIL and a consolidated overview of the approaches in the MBT-area. In the field of model based testing, Utting et al. [116] present a taxonomy of MBT-approaches. Utting et al. distinguish three main categories, which are *Model Specification*, *Test Generation* and *Test Execution*. The model specification aspect of MBT-approaches describes an abstracted form of the underlying rationale of the SUT. This comprises the type of the data expected that is passed to the SUT and the expected output, the semantic / mathematical underpinning of the model itself and the used paradigm for analyzing the produced data of the SUT. In the context of Utting et al.'s taxonomy, APRIL utilizes some MBT aspects. Here APRIL models discrete model artifacts (e.g. Classes, Attributes, etc.) of which the objects states are recorded to be analyzed as history data output that is typically checked offline, which means that the

SUT runs before the test suite. In Utting et al.'s *Test Generation* category, the APRIL framework makes use of an own language to specify test cases.

The big majority of the presented approaches uses automatic test case generation based on an abstract model of the SUT to investigate the interesting properties and corner cases of the implementation. Creating tests with APRIL does not prescribe a certain way of creating input data to trigger the test cases, the SUT produces its corresponding history output-data from. Nevertheless, it must be guaranteed that the mapping between the test case (trigger), the historical data generated from the SUT and the APRIL business rule is maintained, in order to be able to conclude on the evaluation results. Moreover, the automated creation of test cases as it is aimed by most MBT-approaches, is outside the scope of APRIL. APRIL translates one formal notation of test cases into executable representations, automatically.

2.9.2 Category-Partitioning

Generally spoken, approaches that are in the field of category-partitioning aim at a good test coverage. It is typical that next to investigating if the main use cases of a SUT work correctly, an additional aspect is to cover the corner cases that determine if a system behaves correctly within certain defined boundaries. Ostrand et al. [90] describes how the methodology of partitioning works. In principle, the basic steps are similar to the introductory steps of the methodology presented for APRIL, which is to brig the original natural language specification in a semi formal form. However, the aim is a bit different, which is to assign possible values states to attributes found for rule concepts in order to be able to use a formalism to concatenate them for modelling the systems behaviour. This is also suitable to cover corner cases, for testing the borders in which a SUT behaves correctly and also checking when a SUT leaved these borders, by altering one or many variable states. With the APRIL language, testing the borders is possible by incorporating the valid and invalid behaviour in the business rules itself (e.g. If A then <valid reaction>, If A+1 then <invalid reaction>). However, in this work, the generation

of test cases based on APRIL is a manual step. Moreover, the formalisms that are used in Category-Partitioning approaches are by intention designed to automatically generate test cases, which is also different to the APRIL approach. However, this does not prevent the automatic generation of APRIL-statements at all, which is possible but out of the scope of this work.

2.10 Summary

This section presents an overview of good practice in requirements engineering. Modern software development presents problems and needs that require an active part of requirements engineering to provide consistency and accuracy during the entire software creation process. Hence, this is the motivation for using (semi-) formal languages such as UML class models and OCL for modelling certain types of business rules as constraints of class models. The drawbacks of OCL for modelling behavioural rules has been outlined and the advantages of Tempura in this field have been highlighted, which is considered justification enough for the introduction of Tempura, and consequently ITL, as a semantic basis for these kinds of rules. To support the claim that temporal logic frameworks are suitable for modelling behavioural rules, the abilities of ITL are mapped to the needs imposed by state-of-the-art business-process modelling. This chapter concludes with an overview and discussion of related work.

Chapter 3

The specification of business rules

"Wir wollen alles vermeiden, was Schnörkel und Überladung ist, und Schnörkel heißt mir in einem Buche alles, was nicht Buchstabe oder Interpunktion ist." – Friedrich Schiller in einem Brief an seinen Verleger Cotta

English:

"We want to avoid anything of embellishment and adornment, for, in my book, embellishment means all that is not letter or punctuation." – Friedrich Schiller in a letter to his publisher Cotta

3.1 Introduction

Before the diverse types and specification means of business rules are introduced, the context of some of the important notations to specify business rules is presented. A supporting image is depicted in Figure 3.1. Central aspects are the processing techniques and paradigms that utilize the expressive power of certain notations to specify domain models and business rules. As mentioned in Chapter 2, requirements engineers typically use natural language to specify both models and the related rules. Here, natural language processing can help to transform pure natural language to formal models [50]. However, the mainstream NLP approaches of the last decade are based on statistical techniques that provide only a limited degree of accuracy [17], which can become critical if the processing result is not reviewed by the requirements engineers that are typically not concerned with the formal target languages. Hence it is questionable that potentially not fully understood formalized business rules can really benefit requirements engineering. Nonetheless, this topic is handled in Section 3.2.3 for the sake of completeness when the question indeed arises of how requirements engineering benefits from natural language processing.

Other more pragmatic and rather software development-rooted paradigms like MDSD and the Model-Driven Architecture (MDA) focus on meta-modelling (see 3.2.4) to build Domain-Specific Languages (DSLs) designed to be used by domain experts or so-called general purpose languages typically rooted in logic frameworks (e.g., predicate logic or temporal logic). The idea behind a DSL is to provide a highly focused, easy-to-understand, and computer-processable notation for an exact and clear-cut modelling purpose. Domain-specific languages are built from scratch, which means that the design of the language starts with the definition of "what to describe" in the domain (see Foundation in Figure 3.1) and "how to describe it" (see Notations in Figure 3.1). Hence, a DSL has a formal grammar, a parser, and transformation rules that can be used to implement the compiler or interpreter logic for processing the statements made in that DSL. Unlike DSLs, General Purpose Languages (GPLs), such as OCL, strive to give as much expressive power as possible to the kind of formal model they are designed for. Both

GPLs and DSLs benefit from a suitable tool support that is useful in practice, which has also gained the acceptance amongst practitioners. Section 3.3 presents a brief overview of the state of the tooling practice, taking into account selected features that substantially contribute to the usability of the tools.

The remainder of this chapter is as follows. In Section 3.2, the rationale for using controlled natural language for specifying business rules is given, which shall also help to justify the choice for APRIL. Some problems with pure natural language as compared to controlled natural language are outlined in Section 3.2.1. Section 3.2.3 discusses why a putatively obvious choice, which is to utilize natural language processing for business rules formalization, can become critical. A recourse for this dilemma is presented in Section 3.2.4 with meta-modelling. The state of practice in Section 3.3 supports the importance of the meta-modelling way by showing how mature some tools have become in reflecting their acceptance in the modelling practice. Section 3.2.6 shows how sophisticated modern meta-modelling approaches can get in the attempt to unite syntax and semantics for creating a unified standard for natural language requirements specifications materializing in the Semantics of Business Vocabulary and Business Rules. Before a summary on notation forms in Section 3.2.8 is given, Section 3.2.7 completes the discussion on notation forms by reflecting on the formal notation forms.

3.2 Notations for business rules

The overall structure of a language is the principal issue for learning and remembering its constructs [18]. Moreover, Bowden et al. identify two crucial aspects for orthogonal language design:

- *"Lexicalization: the representation of the underlying semantics of the domain as discrete units of the surface structure"*
- *"Syntax: the ordering of the discrete units in well-formed formulas [...] of the language."*

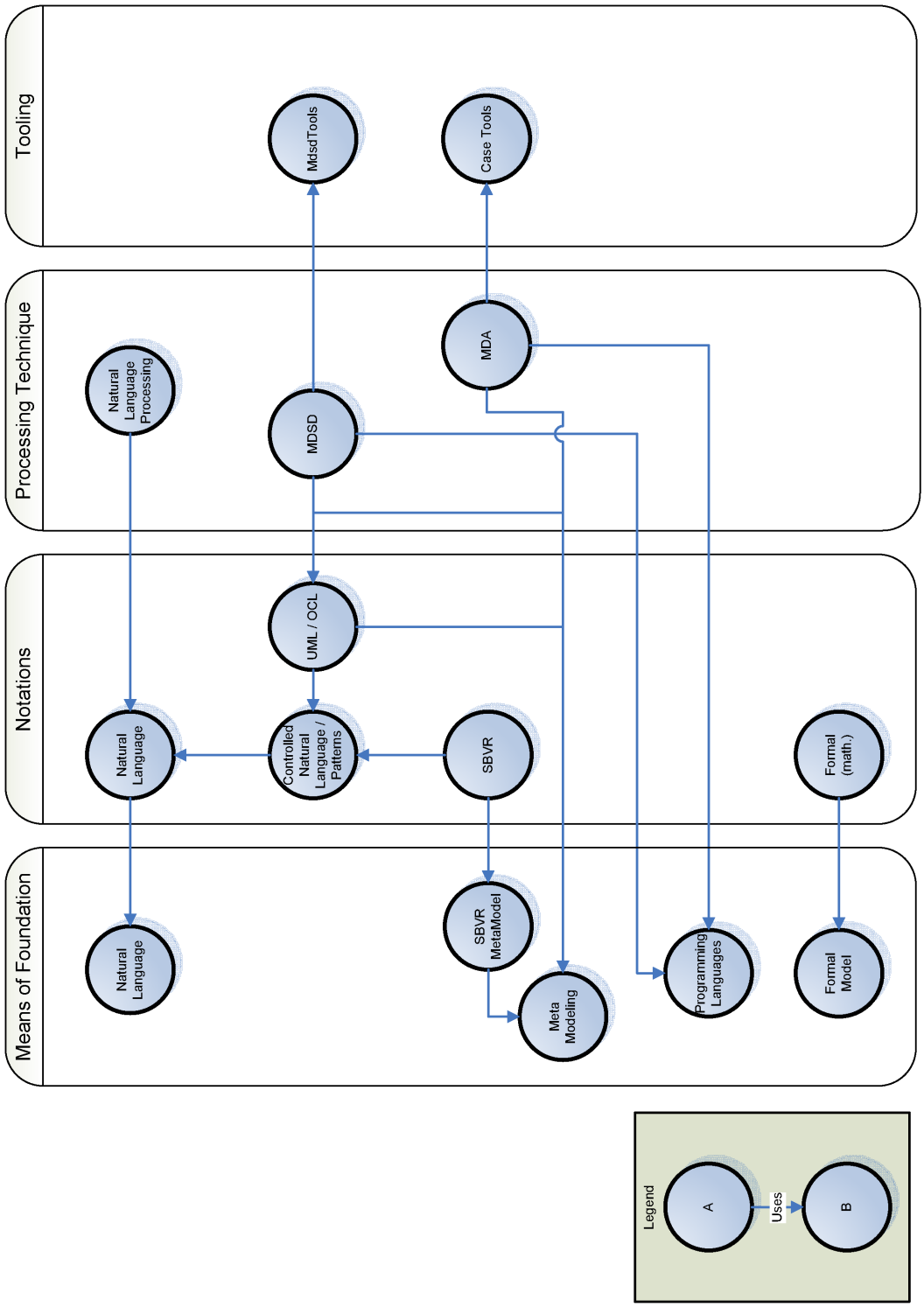


Fig. 3.1: Context of notations and paradigms

A human effort in learning a language consists of the following tasks. First is that the terminology has to be learned. Second is that the congruence of the expressions and their semantics has to be made clear. Therefore, a meaningful and intuitive syntax is necessary. In other words, it must be clear which expressions can be combined to form valid and sensible statements. Therefore, it is helpful to minimize the terminological and syntactic structures. However, the most intuitive notation for the group of stakeholders we are aiming at is natural language. Therefore, APRIL uses a limited set of natural languages for logical expressions and also tries to delegate parts of the language design to the user by permitting the definition of flexible mixfix operators in the so-called APRIL Definitions (see Section 4.2.3). In addition, statements made in APRIL are based on a universe of discourse modelled in a standardised graphical and easy-to-learn graphical notation that requires the user to define domain-specific business concepts that are used as expressions in APRIL-statements. The following sections present a discussion on notation forms using natural language.

3.2.1 Natural language

Specification documents are the initial artefacts when it comes to implementing business logic in IT systems. Specification documents are usually created in cooperation with domain experts and requirements engineers, whereas the technical background of these groups differs tremendously. Requirements engineers fulfil cross-cutting concerns and are negotiators between the business and technical worlds. Therefore, communication is the key between all stakeholders involved. Thus, the most appropriate means to communicate is common natural language. [98] However, natural language specifications cannot be processed by computers to a satisfactory extent per se. Early attempts to transform natural language to formal representations had to tackle the inherent problems in natural language [50]. Nelken et al. [81] mention some problems in the analysis of natural language with regard to logic. Amongst other points they mention the following:

- Ambiguity and imprecision, e.g.:

- Mismanagement of synonyms and homonyms
- Conjunctors and disjunctors may be interpreted in several ways, changing the semantics of a sentence.
- Use of plural to denote sets of entities without stating the exact properties of the sets (e.g., borders, duplicates, etc.)

- Links between sentences, e.g., mismanagement of the use of pronouns
- The temporal structure of specifications
  - Specifications are usually written in a generic present tense or future tense. This makes reasoning difficult for the intended temporal behaviour.
  - Quantifications over events

To overcome those obstacles, two major approaches have surfaced that enable the mapping of natural language to computer language representations. Both are based on a rigorous structuring and limitation of the initial natural language specifications. One approach is the pattern building of natural language clauses (see Section 3.2.2); the other pursues a more generic approach, utilizing grammars [14] (mostly context-free grammars) to define a language's design (see Section 3.2.3.1).

3.2.2 Patterns

Frequently used business rules with the same syntactic morphology can be classified and abstracted to a pattern. A pattern represents a class of sentences that can be instantiated by filling the predefined placeholders, which are usually like typed parameters [98]. Musen et al. [14] describe the means and a methodology for defining mixfix patterns and templates that in turn define a reusable natural language with gaps to fill in. Each pattern is mapped to an artificial language with a precise semantics, which is based on first-order predicate logic. The first step of the methodology is to identify axioms that follow the same structure. Mostly verb phrases with variable subject and direct object constituents are

substituted by typed placeholders. The third step is to derive the generic properties for categorizing the templates. Sosunovas et al. [106] employ a grammar-based approach to enable the definition of textual patterns. The semantic foundation of the production rules of the grammar is based on a meta model that links natural language syntactic structures with logic operators and conceptual schemas based on Object Role Modelling (ORM). The methodology to utilizing the approach presented consists of four steps. The first step is to identify domain-specific business rule patterns quite similar to [98]. This is a creative task yielding informal output and cannot be automated. The next step is a classification of the collected patterns for abstraction purposes, followed by a formal description of the patterns using a pattern grammar elaborated by the authors. This step yields configurable abstract templates. The third step is to bind the templates to the concepts used in the domain of discourse, still preserving placeholders for values that can be filled in in terms of a later instantiation in Step Four. The approaches of [14] and [106] facilitate the process of mapping natural language clauses into computer-processable representations that can either be a formal language or a meta-model. As Section 3.2.4 shows, meta-modelling can be the basis for further transformations.

3.2.3 Natural language processing approaches using grammars

3.2.3.1 Approaches using context-free grammars

Context free grammars (CFGs) [14] are used to describe the abstract syntax of a language. A structured natural language can be described by using CFGs. One approach which utilizes CFGs as the initial step in the formalization process of natural language specifications has been undertaken by [56]. Kleiner et al. present a meta model that links natural language abstract syntax and a semantic model based on SBVR [107], [83] (see Section 3.2.6). Their meta model is a representation of CFG that describes the simplified SBVR version of natural language. The transformation of the abstract syntax of the natural language is specified by QVT-model transformation rules. Query

View Transformation (QVT) [111] is an OMG-standardized OCL-like language that helps to transform Meta Object Facility (MOF)-based models into other models based on MOF. The mapping from SBVR to UML concepts follows a denotational approach using a mapping table. A similar approach by Bajwa et al. [7] also utilizes a simplified version of such a mapping table of natural language constituents into a semantic model based on OCL. Hereby, Bajwa et al. present an approach to translating the structured natural language to OCL via an intermediate SBVR translation. This transformation is divided into four steps. First is to parse the natural language requirements using the natural language syntactic structure [1]. This is aided by an existing tool called Language Engineering System for Semantic Analysis (LESSA). Step Two makes a rule-based mapping of the concrete syntax tree nodes of the parsed text into metamodel elements of SBVR. These rules are generally pretty simple:

- Common names are mapped to noun concepts
- Proper nouns are mapped to individual concepts
- Auxiliary verbs and action verbs are mapped to verbs
- Adjectives are mapped to attributes

Step Three verifies that the identified concepts are available in the UML-Class model which serves as the universe of discourse of the OCL statements to be generated. The verification process actually checks whether the UML-class model and the SBVR based domain model contain corresponding elements of the same name. The corresponding elements are (from SBVR to UML-class models):

- Noun concept and Class name
- Individual concepts and instance names
- Adjectives and attribute names
- Verbs and method names

The final Step Four then applies the semantic foundation which is actually an additional mapping process of the SBVR concepts into OCL statements. Therefore, structural SBVR rules are mapped to OCL header statements constituting the context sets. The actual SBVR semantics holding the logic is cast into the OCL body. In this case, a rule-based denotational transformation is applied, which is based on a syntax mapping table. This work also shows that as long as SBVR lacks proper tool support, e.g., to execute statements, a transformation into better-suited languages like OCL is justified. However, the intermediate step via SBVR may be convenient because of the proximity to natural language structures but it is questionable if this extra step is stable and sufficiently free of errors.

3.2.3.2 CCGs and machine-based learning approaches

Combinatory Categorial Grammars (CCGs) [111] provide more expressive power for describing the abstract syntax of a language. Plainly, this is due to that with CCGs, it is possible to couple semantic knowledge with the syntax. For example, unlike in artificial languages like java, it does indeed matter which side-conditions a certain constituent of a language has to comply with. For example, in linguistics, the grammar rules for English constrain the declension of a verb with regards to the tense being used. There are several linguistic rules that describe the constituents' conjugation or declension mechanisms according to the tense, genus, antilog, and case applicable [1].

CCGs enable the linguistic soundness checking via rule-based inference mechanisms that allow reasoning about the data annotated to a syntaxes' lexical items, which are used to implement those linguistic rules. CCGs are used in machine-based learning methods for textual understanding. In the field of Artificial Intelligence (AI), a list of approaches exists that pursues machine-based learning for mapping natural language into logic formulas [35] [126] [123] [52] [34]. Moreover, it is typical for machine-learning approaches to pursue an open-world assumption with regard to the universe of discourse in use, which becomes enriched and refined after each learning cycle.

In [126], an algorithm is described that automatically induces a context-free grammar that maps natural language sentences to a logical form encoded by a lambda calculus representation. The algorithm uses a probabilistic model to aid the parsing process, which is based on grammars specially designed to deal with natural language. These grammars allow semantic annotations in combination with the abstract syntax (Combinatory Categorial Grammars [50] [123]).

Authors of [35] pursue a similar approach of relying on syntactic-semantic training pairs in Support Vector Machines (SVMs) by means of kernel functions to enhance the probabilistic mapping. This is due to encoding the semantic representation of natural language questions by the Structured Query Language (SQL) [65].

In [123], a statistical approach to semantic parsing is presented. Therefore, a word-based model is used for expression-based learning. In combination with the concrete syntax tree that model can be seen as syntax based transformation model. However, lexical learning depends on a lexicon, which is also learned and not predefined making the entire process by far non-trivial and prone to errors.

In [52], a kernel-based approach for learning semantic parser is presented. Therefore, Support Vector Machine (SVM) [21] [112] classifiers based on string subsequence kernels are trained in the meaning representation language for each rule of the abstract syntax. The trained classifiers are then used to build complete meaning representations of natural-language sentences.

The authors of [34] equip natural language parsers with certain heuristics and learning algorithms which enable annotating a concrete syntax tree of a natural-language sentence with semantic elements. They are therewith allowed to generate a formal meaning representation. However, a feedback mechanism for confirming a correctly matched tree-node annotation pair then has to be utilized.

All of these approaches are based on probabilistic heuristics which introduce a non-neglectable degree of uncertainty in the mapping process between natural language and formal semantics. Another phenomenon that machine-learning methods suffer from is

over-fitting if not handled correctly [97].

Nevertheless, methods using the approaches presented may be useful when working on existing large requirements documents and tool support is needed for making suggestions on, e.g., which business concept should be derived from a term.

3.2.4 Meta-modelling

Generally, a metamodel is a model that describes the constituents of other models based on the metamodel. The idea is to move away from domain specific ontologies into a higher "meta" layer. Based on the meta layer, certain logic can be defined that enables tasks like reasoning or transformations into other models. The paradigms presented in the following subsections utilize meta-modelling to substantially generate software artefacts from models, aiding the development process.

3.2.5 Model-driven software development

Work in the context of Model-Driven Software Development (MDSD) focuses on integrating domain experts and requirements engineers into the processes of software development, software maintenance, and software evolution [109]. This is mainly supported by models like those provided in the UML-specification [84]. Nevertheless, strong emphasis is laid on flexibility in building tailor-made domain-specific modelling languages and thus provide a suitable support for the specification process. Domain-Specific Languages (DSLs) can either be textual or graphical. A considerable tool landscape exists for defining DSLs (e.g. [48, 3, 49, 69]).

3.2.5.1 Meta Object Facility

Meta Object Facility (MOF) [87] is a standardized Meta-Meta-Model specification by the Object Management Group. It is usually applied in the sector of model-driven engineering . The modelling architecture of MOF is depicted in Figure 3.2. It describes a closed meta-modelling architecture, which is divided into four layers, M0 to M3. The layer M0 holds

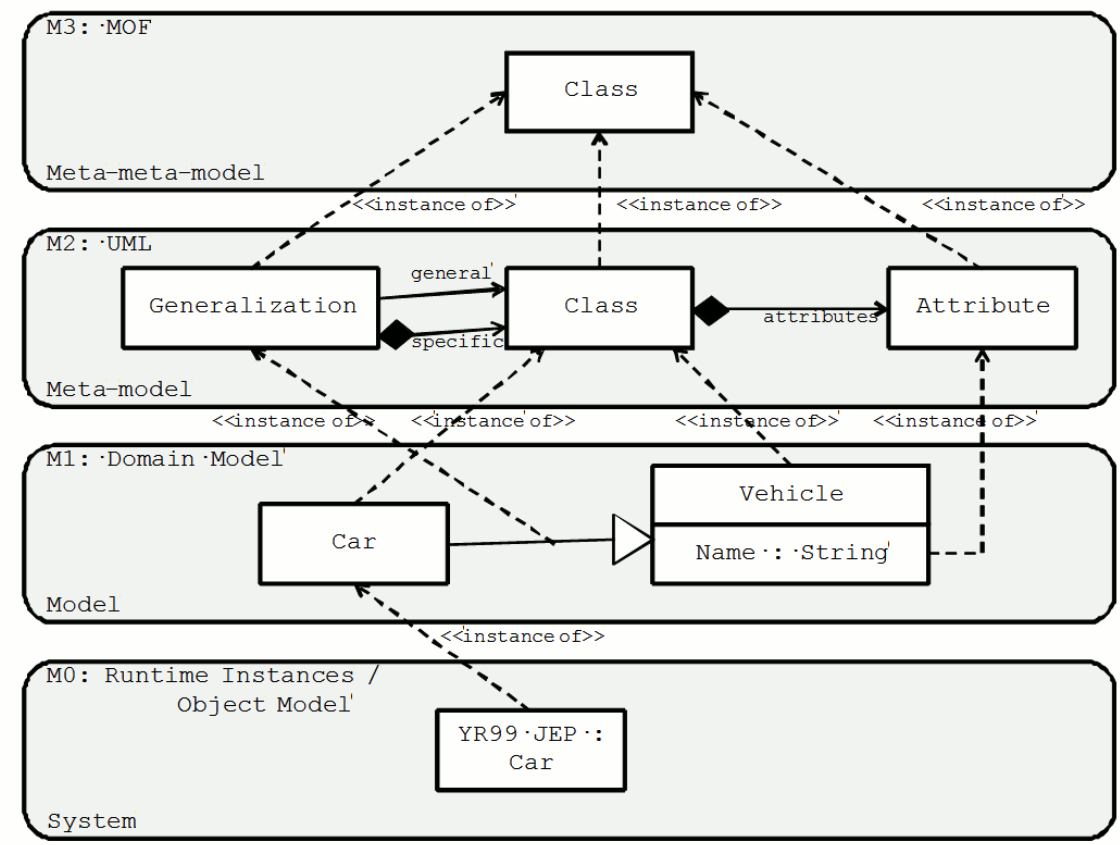


Fig. 3.2: Meta Object Facility (MOF)

real world objects, whereas each higher layer is the abstraction of the layer below. In contrary, each notation element of layer n-1 strictly corresponds to its abstraction element in layer n. Layer M3 is compliant in itself, determining the closed character of MOF.

MOF is utilized for defining data structures, which can also manifest in abstract syntax trees for Domain Specific Languages. EBNF and MOF are intentionally the same with regard to their domain.

3.2.5.2 Unified Modelling Language

The Unified Modelling Language [84] by now is a set of 14 standardized mostly graphical modelling languages, e.g., for the specification of conceptual schemas, business processes, system architectures, and other things of the real world. UML is an offspring of the Object Management Group (OMG) and is actively used and steadily evolved by researchers and practitioners. Several model types addressing distinct modelling concerns

are united within the UML. The Object Constraint Language (OCL) (see Paragraph 3.2.5.2.2) is part of the UML. Up until now, APRIL is based on the UML-class model and OCL, both of version 2.3.

3.2.5.2.1 Profiles A UML profile [100] [88] is a user-defined extension mechanism for user-defined DSL construction. This is due to the demand of the UML-using community to be able to extend UML's standard constructs with domain-specific notation elements. Therefore, certain extension points are defined within the UML metamodel itself. This is done with the so-called stereotype, which is actually a metamodel element that can be used as an annotation to a class-name. Stereotype annotations are wrapped in guillemets. A class C in a profile definition, of which the name carries a «Stereotype» annotation, is a stereotype on its own when used in the domain model. The extension mechanism constrains each with the «C» -stereotyped class Dn of the domain model to be extended by the attributes and methods of class C. The implementation of the extension is done via annotations attached to the class Dn. A generator that understands the logic of its dedicated profile is then able to process the parts of the domain model it is concerned with.

3.2.5.2.2 Object Constraint Language The Object Constraint Language [84], [114] (OCL) is a formal, declarative language used to constrain or extend UML-class models for implementing business rules. It is based on a first-order predicate logic and operations on sets. Its syntax is similar to Smalltalk. The basic idea that OCL pursues towards the definition of business rules is to specify invariant, pre- and post-condition rules on an object model based on a UML-class model. Pre- and post-condition rules can only be defined on methods of classes. The predicate logic statements are then defined by sets of objects or properties of individual entities or a combination of both. Thus, for example, a filter may select certain objects being a result of a Cartesian product, and a preceding function then applies a logic operation which can then be evaluated as true or false when applied to the filtered set. As with any logic based on propositional calculus, invariant,

pre- or post-conditions can only yield either true or false values. However, there are implementations of OCL that support a tri-state logic [114], in which the third value is denoted as undefined. A small impression on how complex OCL-constraints can become is given in Listing D.0.3. OCL is part of the UML Specification.

3.2.5.3 Model-Driven Architecture

The Model-Driven Architecture MDA can be seen as a methodological implementation of the MDSD paradigm. MDA defines different viewpoints on data or structures (e.g. business rules, business processes, or business architectures). Therefore, different levels of abstraction are defined in the MDA, which provide each group of stakeholders with the data they are typically concerned with. The following layers are usually mentioned in the context of MDA.

- Computation-Independent Model (CIM): This most abstract layer is dedicated to domain experts and requirements engineers and incorporates natural language, tables, images, and any arbitrary form of notation, which are mostly informal.
- Platform-Independent Model (PIM): This abstraction layer is dedicated to requirements engineers and incorporates structured natural language, models, and semi-formal notations.
- Platform-Specific Model (PSM): This detailed layer is dedicated to technical staff, programming languages, configuration artefacts, and others.

MDA's understanding of a platform is a set of subsystems and technologies providing a coherent set of functions, interfaces, and use-case patterns [33]. Depending on the viewpoint, each model in a certain layer hides or enriches the data of the layer lower than the above. The step from a more abstract layer to the related detailed layer is achieved by (model) transformations (see Section 3.2.5.4.2). The basis of each model and transformation in the MDA is an appropriate domain-specific language (DSL). With UML Profiles, it is possible to define own Domain-Specific Languages based on the core

set of UML's notation elements (see Section 3.2.5.2.1). Another aspect addressed by the MDA is the capability of a seamless data exchange between different specifications or MSDS/MDA-tools. Therefore, the Extensible Markup Language (XML)-based XML Metadata Interchange (XMI) has been published.

3.2.5.4 Paradigms and techniques in Model-Driven Software Development

This section presents the paradigms and techniques in the Model-Driven Software Development.

3.2.5.4.1 Abstraction and separation of concerns Model-Driven Software Development (MDSD) aims at supporting the best practices of software engineering, especially Separation of Concerns (SoC) and the reusability aspects achieved by the abstraction of business logic as well as their representation elements. These two core aspects can be seen as orthogonal dimensions. SoC [25] helps to ensure distributing the concerns amongst different stakeholders, e.g., a software architect will manage the morphology of the software application itself whereas a domain expert will deal with the business rules. This is supported by techniques like, for example, the Aspect-Oriented Programming (AOP) [101]. Figure 3.3 gives a short and by far not an exhaustive overview of the different related fields.

The elements incorporated in Figure 3.3 can be described as follows:

- Each pyramid layer denotes a different level of abstraction, where the level of abstraction decreases from the top to bottom, while at the same time, the level of detail increases.

- The terms in the blue-highlighted cross-cutting concern, tabs show the aspects of cross-cutting concerns which are related to a certain concern in each abstraction layer.

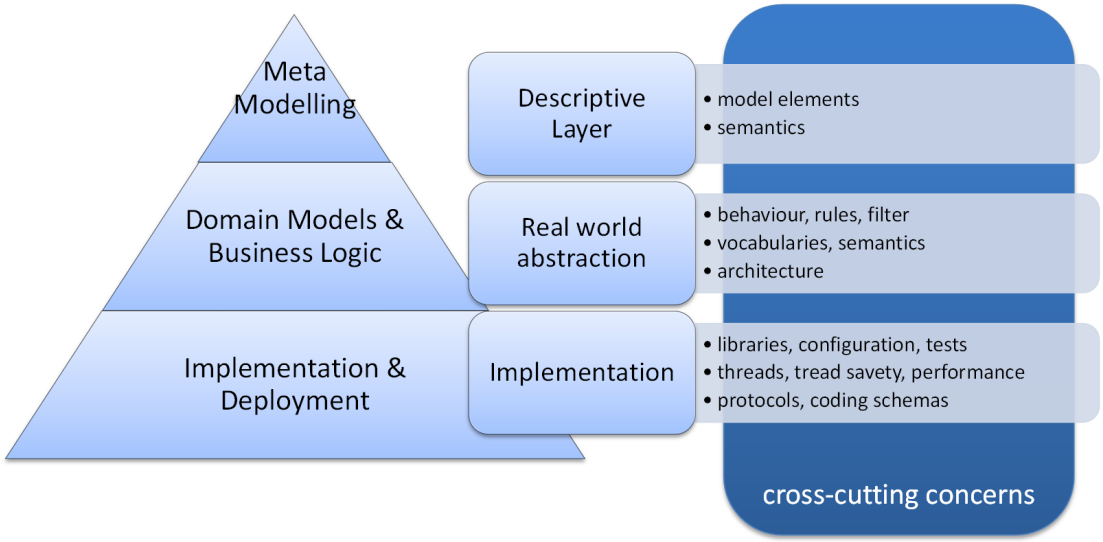


Fig. 3.3: Abstraction and separation of concerns in MDSD

3.2.5.4.2 Model transformations In MDSD, model transformations are the glue between any kinds of related models, either on the same or on different abstraction levels. In general, a model transformation is nothing else than a compile task. In practice, model transformations are distinguished between Model-to-Model (M2M)-transformations and Model-to-Text (M2T)-transformations. The term M2M transformation is normally used when specifications that are typically graphical are cast into other graphical notations. On the other hand, M2T is the typical model-based code generation. Established techniques for achieving an M2M transformation have been employed by transformation languages like OMG's QVT [111]. Whereas, M2T transformations mostly rely on template engines, e.g., xpand [27]. Therefore, templates are artefacts that encapsulate the transformation logic.

3.2.5.4.3 Templates Templates are the artefacts used in M2T transformations. Templates consist of text blocks which may actually be yet un-interpreted code fragments in the target language and control structures like loops and alternatives as well as common mathematics and string operators. These control structures and operators are applied on metadata as well as on the fixed text blocks to generate text output. The metadata can either be objects (e.g., from an object model) or other kinds of data (e.g., from a

database). An example of an M2T transformation using templates may be that a list of classes attributed to a UML-class model is iterated, while for each attribute, a text sequence in a programming language denoting a public member variable is generated.

3.2.5.4.4 Round-Trip Engineering The Round-Trip Engineering discipline is a combination of the typical forward-engineering and reverse-engineering. In the context of software development, forward-engineering is to create software based on specifications, including models. The reverse-engineering of software code into abstract models is much more difficult because of at least two reasons. The first is that programming languages have more expressive power than abstract modelling languages and not every construct of the program code can be mapped back to the model. The second reason is that if a proper backtracking mechanism is missing (e.g., code annotations added to denote the original model element), an automated backtracking cannot be realized. Nevertheless, the basic idea of combining both engineering disciplines is to keep the semantics of both representations synchronized, ideally using tool support to automate this process.

3.2.5.4.5 Aspect-Oriented Programming The Aspect-Oriented Programming (AOP) [101] is a technique that separates the distinct logical aspects of a program into different disciplines in order to develop them separately. The attempt is to decouple the so-called cross-cutting concerns (e.g., logging, error handling, transactions management) from the actual functionality or the structure of a program. The logic in the aspect-specific artefacts gets mapped to the functional parts at compile time. The benefit of using AOP is to decompose different aspects of software, which then can be developed and managed separately. This makes the overall development of software less error-prone, and in consequence, less time-consuming. The danger of using AOP is towards the comprehensibility for humans, as the traceability of the logic definitions and the calls in the actual program are not obvious, as it gets realized by point-cuts [28], being not that common, and also offering the ability to prevent or redirect any function call. AOP supports the Separation of Concerns paradigm. AspectJ is a popular implementation of

AOP [12, 28].

3.2.5.4.6 Aspect-Oriented Modelling The rather young Aspect-Oriented Modelling (AOM) [33] is a combination of modelling and Aspect Oriented Programming. In AOM, the representation of elements and their relationships are depicted graphically to make the conceptual schemas more comprehensible, and this is typical for graphical modelling in this case. The authors of [33] mention two challenges in this new discipline. The first is to extract the aspectual information from the functional parts of the requirements models while at the same time preserving the traceability of the links as per the specification. The second is to develop the proper means of specifying and reasoning for the aspect-oriented model specifications, that may then be the basis for code generation.

3.2.6 Semantics of business vocabulary and business rules

The Semantics of Business Vocabulary and Business Rules (SBVR) [83] is an MOF-based metamodel standardized by the OMG. SBVR enables giving business rules and domain concepts a precise semantics based on the SBVR meta model and, at the same time, making business rules/vocabulary libraries exchangeable by providing an XMI-based data format [107], [105], [82], [60]. Currently, SBVR is of the version 1.0 (released in 2008). The practical usefulness of this rather new standard is, amongst others, explored by Linehan et al. in [61] and [60]. The abstraction layer in which SBVR is used is the Computation-Independent-Model (CIM) Layer of the MDA paradigm (see also Section 3.2.5.3). The SBVR is itself a metamodel and does not obligate a representation. However, in the standard, there are two proposed textual representations: Structured English and RuleSpeak. SBVR Structured English is used for semantically creating the metamodel concepts, making the specification self-contained. SBVR can be used to give semantics to business vocabulary (terms), which is described by one of the proposed languages mentioned earlier. At this point, SBVR can be used for the same purpose as the UML-Class model can. Kleiner et al. [56] briefly presented a mapping of SBVR

metamodel elements to the meta model of UML. The other field of SBVR is concerned with business rules based on the business vocabulary. Basically, the statements are made in one of the provided languages. Their constituents can be mapped into the SBVR metamodel, which serves as formal semantics (done so by [56]). The expressiveness is limited to verbalized expressions from first order predicate logic and modal logic expressions [105]. However, second order logic in SBVR is possible when restricted to Henkin semantics [61]. Henkin semantics [124] restricts quantifications to a range of sets with known boundaries. Mechanisms for parsing SBVR-controlled languages are not dealt with in the original SBVR-specification. However, in that sector, promising work has been done by Kleiner et al. [56]. The formal semantics of SBVR is grounded in ISO Common Logic [47]. On the capabilities of SBVR, Linehan [61] states that: *"... Potential benefits include rule validation and consistency checking to automatically recognize problems such as conflicting rules and situations where one rule dominates another (i.e. a second rule never has effect because a first rule applies more generally)..."* However, this is due to the fact that SBVR is based on predicate logic. Further, Linehan argues that SBVR is targeted at an experienced group of stakeholders who are able to express themselves in formal logic using Structured English or RuleSpeak. A considerable benefit is achieved from using SBVR in companies with very large vocabulary and rule libraries (e.g. when dealing in terms of contracts and jurisdiction). A hurdle in adopting the new textual and yet unapproved standard is probably the competitive situation with the widely-used UML with its well understandable and intuitive graphic focused notation. Further, Linehan states that: *"... The practical question is whether practitioners will find that the value obtained from a higher degree of abstraction is worth the effort involved in formulating such abstractions. Significant tools development effort will be needed to maximize the value and minimize the effort of SBVR-style modeling. ..."*

However, in the opinion of the author of this work, another hurdle for the swift adoption of SBVR amongst practitioners may arise. As defined in the meta model of SBVR [83], many aspects of language construction are mixed together, which makes

the definition of new domain-specific constructs in SBVR tremendously complex. These aspects are:

- Model constituents for defining an entity model according to the UML-class model.
- Definition of an abstract syntactical structure (Grammar).
- Linguistic characteristics (e.g., whether a concept is a question or not, noun concepts, or even individual concepts)
- Constituents for the specification of semantics based on predicate and model logic.
- The mapping between semantic and syntactic constituents.
- Aspects of a programming language (e.g., variables and type structures)

Despite the outstanding work that has been done to overcome the obstacles imposed by the inherently complex meta model of SBVR (e.g., [15]), it is unclear how practitioners can be enabled to effectively use SVBR models (instances of the meta model) to effectively extend their universe of discourse for a certain domain. Here the problem may be that any concept in a target language (e.g. a programming language like C# or a formal language like Z) must be mapped first to the respective metamodel constituent of SBVR. This forces collaborating stakeholders (e.g. requirements engineers and programmers) to communicate (particularized) with each other in a very complex formalism which they are not familiar with. Hence, Linehan's last statements can only be agreed to that the trade-off between the formulation of such abstractions having an exchangeable format and the effort of therefore investing has to be carefully evaluated for each individual situation. On the other hand, one has to consider that the methods and tools in the compiler-building discipline have experienced great progress in the last decade.

3.2.7 Logic formalisms

The author's understanding of a characteristic formal notation (FN) is that of a common mathematic notation. Formal notations with respect to describing business rules and

business processes are utilized in the field of formal methods, which is gaining some momentum, particularly in requirements specification [124]. Ostensibly, fields where FNs are applied are safety-critical systems, where health or even lives are at stake. Here, FNs are used for proving the correctness of mostly small programs using theorem provers. Moreover, FNs are utilized to express integrity rules using predicate-logic statements and for process rules using CTL* statements. The third field for which FNs are used is the formal foundation of other languages. As the focus of this work is on natural language rather than on FNs, FNs are not considered here.

3.2.8 Summary on notation forms

In this chapter, we have reviewed relevant notation forms and their processing. Generally, we have to distinguish the graphical and textual notations, where each is with their individual strengths and weaknesses. From the viewpoint of the author, graphical notation forms are better for representing discrete (conceptual schema) models where entities are represented by nodes and relations are represented by edges. This is especially congenial for the comprehensibility of large models. Textual models are ahead in representing logic and query statements based on those graphical models. As this work aims at providing a comprehensible and easy-to-learn language, a hybrid approach combining the advantages of both graphical and textual modelling is used. Therefore, the UML-class model is used to specify the universe of discourse, whereas all logical business rule statements are based on a textual representation designed to be a simple form of natural language.

3.3 Modelling tools

In order to draw a picture of the state of practice, a list of modelling tools has been evaluated (for details see Appendix A). The following evaluation of the tools under consideration is by far not exhaustive. However, these tools had considerable momentum at the time when this document was written. Moreover, the characteristics under

comparison are tailored to be sound in terms of APRIL. The following Table 3.1 presents the criteria that have been used for evaluating the Tools.

Characteristic	Description	Obligation
Documentation and tutorials	Practical usability strongly relies on good documentation to help master the given features within an acceptable timeframe.	must
Data-exchange, XMI-conformity	A standardized data-exchange format is obligatory.	must
UML 2.0 conformity	APRIL is based on UML-Class models; thus, they have to be supported to at least UML 2.0-compliance.	must
Supports other graphical notations	Means to implement own notation elements.	can
Intuitive User Interface	Minimizes the time taken to get used to the features of the tool.	should
Code-generation mechanisms	The possibility to define transformation rules for model elements is obligatory. This can be done in several ways, e.g., by the template engine, the transformation rules, or the output print lines in the code.	must

Table 3.1: Characterization of modelling tools

In order to be able to compare the different tools, a simple formula is applied for arriving at a representative value, which is shown as follows: In this formula, M_n denotes the (school) marks that the feature has gained according to the audit of the author. Whereas, G represents the importance of the feature in the context the respective domain (MDSD / MDA). Hence for a feature mark, G works as a weighting factor that reflects the obligations *must*, *can* or *should*. The overall score v for a tool is the sum of the rated marks of the number of features n.

$$v = \sum_{i=0}^{n} M_i * G \quad (3.1)$$

v : value

M_n : mark of the feature gained

G : impact factor

n : number of features

3.3.1 Case tools

Computer-Aided Software Engineering (CASE) is a paradigm to aid the computer-based conception, planning, and documentation of software artefacts. Typically, CASE tools are technical implementations of language specifications for a certain purpose in at least one of the domains mentioned earlier. The most popular language concerning all these areas is the Unified Modelling Language, UML. Authors [33] argue that CASE tools are antecedents to the more open MDSD tools. The difference between MDSD and CASE tools is considered to be their focus, in which CASE-tools are more on specification and documentation purposes rather than on the code generation of the MDSD tools. However, modern CASE tools additionally employ means of code-generation, which blurs the border with MDSD tools. An aspect not considered in this evaluation is that the support for procedure models as APRIL does not explicitly obligate compliance with a certain procedure model.

3.3.2 Summary of CASE Tools

The evaluation of the tools under examination can be seen in the following Table 3.2. In this evaluation some key features that seemed relevant for this work are compared. Therefore a rating between one (feature is poorly addressed) to five (feature is very well addressed) was introduced. Each rating is co notated by an *impact factor*, expressing its importance compared to other features, ranging from one (low importance) to three (high importance). The features under consideration and their importance to APRIL are explained in Table 3.1. Moreover, the overall rating is calculated by the rule in Equation 3.1.

Feature	impact factor	VP UML 7.0 (Visual Paradigm)	Together (Borland)	Magic Draw (NoMagic)
Documentation and tutorials	3	3	4	4
Data exchange, XMI conformity	3	5	5	5
UML 2.x conformity	3	5	5	5
Supports other graphical notations	1	3	5	4
Intuitive user interface	2	4	4	5
Code-generation mechanisms	3	2	4	3
Total		58	67	67

Table 3.2: Monetary evaluation of CASE tools according to Equation 3.1

3.3.3 MDSD-Tools

Model-Driven Software Development is a development paradigm that has gained considerable momentum. Thus, a wide range of development tools addressing MDSD has been established until now. However, to consider each existing tool would be too extensive for this work. A list of tools maintained by the OMG exists under `http://www.omg.org/mda/committed-products.htm`, which can be the basis for a more exhaustive evaluation. The list of potential tools was limited at least by applying the following criterions:

- The code generation is based on UML models (not only class bodies but also parts of the functionality, e.g., from UML-activity diagrams)
- The data exchange format is compliant to the XMI-standard.
- The capability exists to add or extend the notation elements and create a user-defined Domain-Specific Language.

3.3.4 Summary of MDSD tools

Feature	Impact factor	Andro MDA	oAW	ObjectIF	Meta Edit+	MPS
Documentation and tutorials	3	4	4	4	4	4
Data exchange, XMI conformity	3	5	5	5	5	5
UML 2.x conformity	3	5	5	5	5	5
Supports other graphical notations	1	3	5	0	5	0
Intuitive user interface	2	0	4	5	5	5
Code-generation mechanisms	3	2	4	3	3	3
Total		53	67	61	61	61

Table 3.3: Monetary evaluation of MDSD tools according to Equation 3.1

3.3.5 Conclusion of the Modelling Tool Evaluation

The preceding sections give a brief overview of the MDSD- and MDA-tool landscape along with an evaluation of their capabilities, showing that an exact one-on-one comparison is quite complicated. This is due to the fact that different concepts of the MDSD paradigm are weighted differently, which manifests in the heterogeneous maturity of the tools' functionalities. However, some tools look promising with regard to the ability to aiding a straightforward implementation process, but a solution where domain experts along with requirements engineers can be involved to a satisfying extent is missing. Nevertheless, textual modelling environments are considerably gaining momentum for the user, the developer, and also the scientific communities. That is due to that tailor-made textual Domain-Specific Languages are widely preferred for representing business concepts and business logic. From the viewpoint of the author, the most significant usability towards representing business concepts and business logic is gained by combining the graphical world and the textual world. In consequence, business concepts (conceptual schemas) get specified by graphs, and the related logic gets specified in the common textual way. These shall both be sufficiently formal to contribute to the vision of the Model-Driven Software Development paradigm, which is automated code generation.

3.4 Summary

This section starts with an overview of the existing software modelling notations and semantic frameworks that have evolved over the last decades. This investigation comprises a wide range of frameworks, from the purely natural language over the semi-formal to the formal ones, to describe business rules. Also, natural language processing approaches using artificial intelligence have been investigated. Each approach is discussed in the scope of its suitability to the needs imposed by state-of-the-art requirements engineering, presented in Chapter 2. Modelling approaches materializing

in costly industrial tool support are considered to impose an outstanding role in this field. Therefore, an overview is presented of tools used in practice and dedicated to the paradigms MDA and MDSD. The investigation of the notations, tools and the modelling paradigms serves as the basis for revealing the gap that the new APRIL framework presented in this work is supposed to fill. Roughly, the gap is to bring understandability to formal modelling approaches. Therefore, a part of the APRIL framework is a novel, controlled natural language that uses compiler building mechanisms for a tailor-made support of a formal modelling rationale that can be described by OCL and Tempura.

As discussed in the preceding sections, natural language processing is not being considered, as it imposes probabilistic characteristics not suitable for a reliable requirements engineering. However, NLP can be interesting for certain human-guided pre-steps to facilitate a requirements engineer's initial work, but this is not in the scope of this work, either.

Chapter 4

APRIL: A notation for specifying business rules

4.1 Introduction

Natural language specifications suffer from ambiguity, incorrectness, and often non-uniformity of linguistic expressions. A tendency to emphasis those effects arises when more people cooperate in creating a specification, and each individual contributes his/her personal linguistic flavour. In software engineering, an ambiguous specification can lead to divergence between the intended solution and the implemented solution that originally should have solved a certain problem. This, in consequence, leads to a reduced efficiency in the software creation process. Surely, there may be other aspects than the mentioned, causing quality flaws in natural language specifications, which are covered by different researchers. Thus, the justification for this work, seeking to remedy these flaws, should be presented appropriately. For decades, many state-of-the-art techniques addressing unambiguity in specifications have influenced the evolution of the practical disciplines. In Chapter 3, a number of state-of-practice approaches are presented, from which the UML is chosen as a basis for this work. Business rules are specified as additional logical statements which restrict typed sets based on UML-class models, describing a universe of discourse for a certain domain. The use of such techniques (UML), combined with formally definable business rules in requirements specifications, paves the way for modern paradigms like the Model-Driven Software Development (MDSD) to contribute to the overall efficiency of software engineering. The intention of this work is to show how test code can be automatically created from formal business rule specifications verbalized in APRIL by means of MDSD. The test is suitable for checking automatically whether the implementation diverges from the specification. Both the automatic testing as well as the more exact specification are considered to boost efficiency in the software creation process.

One reason for the existence of a novel language (Adaptive Process and Business Rule Integration Language, in short APRIL) for specifying business rules is that standardized formal languages like OCL quickly tend to become cryptic [16]. Natural language, on the other hand, would be sufficiently clear, but it is not suitable, either. Consider

Costal [20] and Halpin [40], who argue that there is a need to substitute the pure natural language business rule notation with a formal language which is non-ambiguous and able to contribute to MDSD methods. The other side of the medal is that the benefits that arise from using artificial languages have been brought about at the cost of clarity, thus precluding any non-technical domain experts from the software creation process [43] if this language is too artificial or mathematical. Further, handcrafting business rules in a so-called formal textual general purpose language like OCL is time consuming and error prone as the user has to express the semantics of the business rules consisting of statements, intricately composed of general operators (e.g., universal or existential quantification). However, this work is motivated to provide a clear, formal language that provides syntax and style rules that guide the specification of business rules that provide better understandability inherently arising from the textual business rule itself, which supports the usability and maintainability by non-technical people. Thus APRIL, which is a formal declarative language for business rules, is presented in this chapter. APRIL's syntax is based on natural language concepts that derive from logic frameworks and operations on sets as well as common business constraints [71]. The aim is to enhance UML class models with additional semantics that can be understood by domain experts with weak technical background. It shall be shown, that this approach is suitable to address a broader reviewer audience for formalized business rules, especially amongst business people, which can lead to a reduction of design errors in a specification.

The semantics of APRIL is based on OCL and Tempura. APRIL statements are expressed in structured natural language and can be compiled into OCL and Tempura expressions, depending on the type of business rule. Then they can be processed further by existing engines, e.g., the USE Tool, Dresden OCL Toolkit, or Borland Together [115, 11, 36]. The aim of this work should be to achieve better understandability in the process of formal requirements engineering using concepts known from other formal languages combined with novel concepts based on natural language. First is the concept of decomposing the business rules into simpler entities. The means for this are package

building, the easy and clear definition of sub-expressions (called Definitions in APRIL), and intra-rule variables. Second, by allowing the use of mixfix notation at certain points, it is possible to verbalize statements that are pretty close to natural language. Third, we use a type system that is similar to that of the target languages, which supports APRIL's static typing, making it easier to conduct semantic checks.

4.2 Specification

This subsection presents an overview of the features of APRIL, which allow understandably verbalizing business rules that can be tailor-made to fit a particular target domain. First, the rationale of APRIL is explained, followed by an introductory example to demonstrate how close to natural language APRIL statements can indeed be. Starting with the example rule, the enabling APRIL definitions and atomic formulas are explained step by step. The subsection concludes with a threefold description of the translation of the atomic formulae into the target language, a presentation on how to introduce custom atomic formulae into the language, and an indication of the use of common constraints within APRIL.

4.2.1 Business rules in APRIL

In general, the different types of business rules in the industrial practice are: Integrity Rules, Derivation Rules, and Rules to describe behaviour [15]. Despite that there are fundamental intentional differences, these rule types have one aspect in common: the description of the semantics of parts of the real world into formal representations by means of logic. In APRIL, we use UML-class models [84] to formally represent the concepts of a business domain. The reason is that the UML-class model is widely used, well understood, and has undergone a prolonged period of mellowing, which makes it particularly suitable for representing conceptual schemas. Own investigations (see Chapter 8) show that UML-class models can be understood by untrained persons in a very short time and even without preparation, at least to a sufficient degree. APRIL requires UML-class models as the domain of discourse to specify business rules as constraints. These constraints are of the following types: **invariant, pre- and post-condition, and behavioural rules**. Invariants describe allowed system states that must not be violated at any point in time. This is unlike the pre- and post-conditions, which are restricted in scope right before and after a transition. The fourth rule type describes behaviour explicitly.

Behavioural rules can describe operations lasting over multiple state transitions [119], which is not possible with a single pair of pre- and post-conditions.

Choosing which kind of APRIL business rule type to apply (invariant, pre-, post-condition or behavioural rule) depends on the problem that shall be described. For describing a single state transition, it may be reasonable to use a pre-, post condition rather than a behavioural rule, imposing the application of a different logical framework. Therefore, behavioural rules get translated to Tempura, whereas pre-, post-conditions are translated to OCL. Tempura allows describing behaviour in a much more natural and finer grained way than OCL does. Moreover, Tempura also allows the application of the ISEPI Method, which is explained later. Once a suitable business rule type is chosen, the syntax for the APRIL-language offers the appropriate introduction statement, making the types easy to distinguish for a reader. Business rules introduced by the key word "*Behaviour*" get translated to Tempura, all other APRIL rule introduction statements lead to a translation into OCL.

Form the viewpoint of the semantics of APRIL rules, it is required to evaluate a business rule either to hold or not[1]. This means that the return type of the used statements must be of type *Boolean* (the set of possible return value is *true* | *false*). Hence, if in a business rule (no matter if invariant, pre, post- condition or behavioural) contains an operation yielding a different return type other than *Boolean* (e.g. a Set) the corresponding return value can only be used as a parameter for an operation yielding a *Boolean* value at the topmost level.

In order to give the reader an impression of what APRIL can do, an example from the logistics domain is presented in Section 4.2.2. The aim is to show (1) how closely related to natural language APRIL indeed is, (2) to present a brief review of how APRIL is used, and (3) how it is possible to expand APRIL's expressiveness by integrating hand-crafted atomic formulas into the framework.

[1] enforced by the production rules for boolean expressions in the APRIL grammar

4.2.2 Introduction of an example order system

Now, a small, artificial example is introduced of an order system conceived to demonstrate how APRIL can be applied in a typical business rule context. This example is based on the class model in Figure 4.1 and contains enough concepts to be suitable for a range of other example use cases.

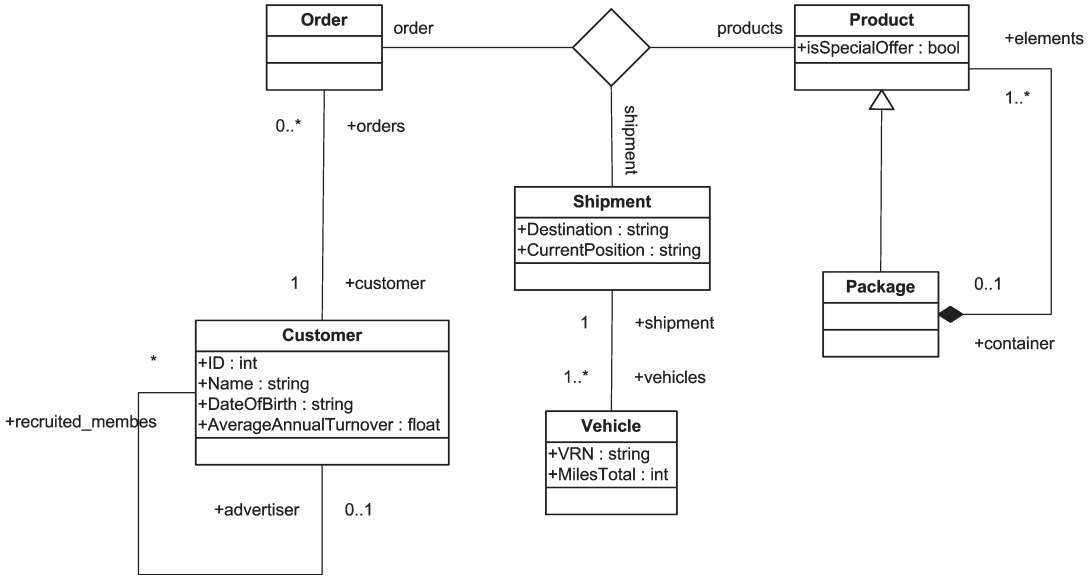


Fig. 4.1: UML model of the example domain model.

In Figure 4.1, a simple domain model of an order system with the basic concepts *Order*, *Customer*, *Shipment*, *Vehicle*, and *Product* is modelled as a UML-class model. As an example of using APRIL on the class model, the corresponding statement for the invariant *rule1* can be seen in Listing 4.1.

Line 1	***Invariant** rule1 **concerns** Customer **is defined as***
Line 2	*All premium customers who order a special offer must pay*
Line 3	*0 EURO for the shipment of that order.*

Listing 4.1: Top-level rule, composed of several APRIL definitions.

The header (Line 1) of a rule contains its name *(rule1)* and the token after the keyword **concerns**, which represents the context set (represented by the class name *Customer*) of the business rule to which the formula after the colon applies. With respect to UML models, the context for invariant rules and for pre- and post-conditions is represented by a

class name and a qualified method name, respectively. The rule body (Lines 2-3) contains the actual business rule. In order to use a natural language sentence in the formal way needed, a few definitions have to be installed, which is explained in Section 4.2.3 directly after this example. Moreover, a detailed specification of APRIL includes default logic and set operators as given in the Sections 4.6.1 and 4.6.7.

4.2.3 APRIL definitions

APRIL definitions are special mixfix operators which allow the intuitive construction of patterns that decompose large business rules into smaller, comprehensible, and reusable sub-statements. Mixfix is a technique particularly useful in forming natural language statements [43]. Mixfix operators allow composing an operator's constants and placeholders in an arbitrary order. The design of the APRIL definition headers is based on the sequence of static name parts and placeholders. Both static name parts and placeholders can be composed arbitrarily to express a business statement reflected as a natural language sentence pattern. Consider the following non-formal schema for exemplification of a definition signature, in which "CONSTANT" denotes a constant in the context of the pattern and (PARAMETER as TYPE) constitutes a placeholder definition with a type prescription. The pattern is placed between the ***Definition*** and the ***yielding*** keyword.

Definition CONSTANT1 (PARAMETER1 **as** TYPE) CONSTANT2 (PARAMETER2 **as** TYPE) ... **yielding** ...

Pattern built this way make it particularly easy to be construct by humans [98]. The example definition in D.1 of Listing 4.2 shows a definition signature between the *Definition* and the *yielding* keyword. Here, placeholders are wrapped by the outermost brackets, and keywords are mixed together to constitute a pattern. Due to the use of the mixfix notation, sentences based on the pattern come close to resembling a natural language sentence when the placeholders are filled in with the correctly typed concepts of

the domain model or other definition calls embedded in the placeholder. The concepts of the domain model represented as UML classes are used for typing placeholders, imposing a restriction on that placeholder, which says it can only be replaced by a statement that is compliant with that type.

Despite the convenience that mixfix operators provide for humans, it is challenging to implement the parser logic [22], especially for nested definition calls (detailed disquisition in Section 5.2). The problem is that the parser has to recognize a *definition call* embedded inside an ID-token sequence in what is, in the grammar specification, another *definition call* (see highlighted EBNF-grammar rules in Listing 4.3). As a consequence, a conventional context-free grammar provides only insufficient means to specify sub-ID-token streams with different semantics for their embedding of ID-token streams. To overcome this, the ANTLR v3 [92] parser/compiler-generator framework is used. The framework allows specifying semantic annotations [91], which is actually user-defined code (e.g., in C# or Java), that gets inserted into the proper positions of the grammar to guide parser decisions based on the semantics of tokens. Consider Listing 4.3, where the Boolean return values of the semantic annotations indicated by α_0 and α_1 influence the resolution algorithm generated by the parser. The semantic annotations indicated by the symbols α_n represent code that gets integrated into the parser. The implemented logic provides the link between syntax and semantics. For instance, when a token with the value *Customer* gets recognized, the semantic annotation allows concluding on further decision steps for the parser.

Considering Listing 4.2, then in (D.1) the orders of specific customers are mapped to a shipping price. On the other hand, (D.2) is a set comprehension on the set of all customers, defining what a premium customer is. Furthermore, (D.3) defines attributes that characterize special offers. Given the example from the previous sections, the APRIL Definitions (D.1)-(D.3) of Listing 4.2 decompose the business rule statement from Listing 4.1 into reusable and easy-to-define sub-statements with a signature in mixfix notation.

(D.1) ***Definition*** *All (customers* ***as*** *Collection of Customer) who order (products* ***as*** *Collection of Product) must pay (price* ***as*** *Number) EURO for the shipment of that order* ***yielding*** *Boolean*
is defined as
every *customer* ***satisfies that every*** *"ordered product"* ***satisfies that*** *shipment.fee = price*
with
"ordered products" (orderer as Customer) ***is defined as each*** *product* ***where*** *product.order.customer = orderer.*

(D.2) ***Definition*** *premium customers* ***yielding*** *Collection of Customer*
is defined as
each *customer* ***in all instances of*** *Customer* ***where*** *customer.AverageAnnualTurnover > 20,000 .*

(D.3) ***Definition*** *special offer* ***yielding*** *Collection(Product)*
is defined as
each *product* ***in all instances of*** *Product* ***where*** *product.IsSpecialOffer.*

Listing 4.2: Rule parts (D.1 - D.3) decomposed as APRIL definitions

```
definition::= 'Definition' nameSignature 'yielding'
              typeDef 'is defined as' ruleBody '.'
nameSignature::= (ID | parameterDef)+
parameterDef::= '(' name=ID 'as' type=ID ')';
typeDef::= ID | ID '(' typeDef ')';
ruleBody::= statement+ ;
statement::= ... | referenceOrDefinitionCall | ...;
referenceOrDefinitionCall::= {α0}modelReference
                           |{α1} definitionCall | ...;
definitionCall::= ID (ID | referenceOrDefinitionCall)* ;
```

Listing 4.3: Grammar snippet for APRIL definitions

4.2.4 Atomic formulas

To provide precise semantics for the definitions and rules, APRIL **atomic formulas** are used. These are verbalized versions of operations on sets, predicate logic formulae, and special common constraints sketched by Halpin [43]. The definition of new atomic formulae is also possible. For example, the *every-satisfies-that* statement of Definition (D.1 in Listing 4.2) is an atomic formula in APRIL that constitutes a universal quantification that is by default incorporated into the language. Some more useful operators are described in the Sections 4.6.7 and 4.5 (also [5]). The default atomic

formulae are for maintaining sufficient expressive power and straightforward translation in the executable representations. Moreover, the Sections 4.4.1, 4.4.2, and 4.4.3 (also [6]) present some syntactical rules for the default atomic formulae to make their syntax and the resulting statements more natural. This is the case, for example, in the auto-mapping of plural to singular symbols. Similarly, in D.1, the symbol "*ordered products*" represents a collection of objects of type Product and the symbol "*ordered product*", which is used as iterator symbol for the universal quantification. One auto-mapping rule states that if any iterator symbol postfixed with an "s" equals a symbol that is in the scope of the same function or definition, then the short form, omitting the "in Collection(<Type>)" declarator, can be used. This only applies if the types can be resolved and the symbol is unique in the entire scope stack.

In order to resolve symbols from their usage to their definition, APRIL uses different scope levels, e.g., the global and local variables known from the most programming languages. The precedence for resolving the symbols is as follows:

- Atomic formulas (with iterator(s))
- Local variable / local method symbols
- Definition signatures (symbolic name with types of parameters)
- Class names and role names from the UML model used in the rule header after the *concerns* keyword

Consider the example definition D.1 of Listing 4.2 containing two nested universal quantification operators $\forall_1.customer(i_1|P(i_1))$ with $P(i_1) := \forall_2.f_{LM}(i_1)(i_2|P(i_2)))$ and $f_{LM}(i_1)$:= "ordered product"(i_1) with iterator variables i_n, bound to the respective universal quantification operator. Note that the universal quantifiers are marked with indexes, which makes it easier to refer to them later. The following annotated and grouped excerpt of D.1 illustrates the earlier definition: In this case, the iterators i_1 and i_2 are related by $i_2 \in Ret_1 := f_{LM}(i_1)$, whereas $f_{LM}(i_1)$ is actually defined as a function

every$_{\forall_1}$ customer **satisfies that** (
every$_{\forall_2}$ "ordered product" **satisfies that** (shipment.fee = price)$_{P_2}$)$_{P_1}$

in the local members (LM) section of definition D.1 after the *with* keyword. Ret_1 is the return value yielded by f_{LM}. This is the local method f_{LM} with the symbolic name "*ordered products*", which takes a single parameter of type *Customer*. The method itself is implicitly typed as a collection by the set comprehension function used in the proposition $P_{f_{LM}}$ of its body, which is: The inference mechanism of the typing of f_{LM} works as

"ordered products" (orderer as Customer) **is defined as**
(**each** product **where** product.order.customer = orderer)$_{P_{f_{LM}}}$.

described earlier by simply adding an "s"-postfix to the iterator so the short form of the operator "***each*** *product* ***where*** ..." can be resolved to the conventional form "***each*** *product* ***in*** *products* ***where*** ...". The symbol *products* can be resolved within the scope of D.1 as this is one of the parameters of type "Collection(Product)". Thus, the set comprehension also yields the same type. If we memorize $\forall_1$ that uses the symbol "*ordered product*" as iterator symbol and we also apply the "s"-postfix mechanism, then "ordered products" is resolvable as a local method. As $\forall_2$ is nested in $\forall_1$, which uses an iterator variable (i_1) represented by symbol *customer* of type Customer, the call to f_{LM} does not necessitate explicitly stating the parameter, which would be the iterator of the surrounding operator i_1 of $\forall_1$. This abbreviation mechanism is similar to that used in λ-expressions in C# 4.0 for method groups. Resuming the body statement of $\forall_2$, the scope stack now adds the symbol "ordered product", which is indicated by i_2. Thus, both the immediate short navigation *shipment.fee* and the conventional navigation "*ordered product*".*shipment.fee* are both valid in this context. The short form for our example is chosen. IT is obvious to see that D.1 has undergone some fundamental refactoring, applying a list of abbreviation rules. It is not incongruous if the reader may have the impression that without these rules, the syntax of composed default atomic formulas (such as the $\forall$-operator "**every** .. **in**

.. **satisfies that** ..") is much more artificial. Constructing the example definition D.1 by using default atomic formulas without applying syntactic abbreviation mechanisms would cause the expression body of D.1 to look like the following:

```
...
is defined as
   every customer in customers satisfies that
      every "ordered product" in "ordered products"(customer) satisfies that
         "ordered product".shipment.fee = price
   with
      "ordered products" (orderer as Customer) is defined as
         each product in products where product.order.customer = orderer.
```

This is considered to be a fairly good example to illustrate how difficult it is to form a well-balanced statement that is not too far away from linguistic grammar rules but fully compliant with the formal grammar of APRIL. As mentioned forming statements like this may impose a considerable amount of creativity on the requirements engineer. However, APRIL is formal enough to be understood by computers, and statements have to be unambiguous, which is supported by the incorporation of a type system. The ability to unambiguously resolve the types of symbols used is obligatory to detect trivial typing faults during design time of a business rule. Moreover, it is helpful in the translation process into the target language as it gives at least some evidence that the business rule is formally correct. Behavioural rules are translated into an executable subset of Interval Temporal Logic called Tempura [74], which is briefly explained later. In order to extend APRIL's expressiveness over general-purpose operators provided by OCL, it is possible to customize atomic formulas and thus tailor the language to a certain domain. A key aspect of APRIL is to delegate the natural language design of mixfix operators and new atomic formulae to the human user, who is considered to be the best choice for this very creative task. How the extension of APRIL with new atomic formulae is achieved is explained in detail in Section 4.2.6 (also in [6]).

4.2.5 A translation example to OCL

Most of the basic operators (atomic formulas) of APRIL (e.g., select-function, see Section 4.6.7) can be translated directly to OCL and do not need any further processing with respect to the APRIL definitions (see Section 4.2.3). Here the translation of the APRIL *AllInstances* atomic formula stands as an example for the translation of basic operators from APRIL to a target language.

$$[\!["all\ instances\ of" < className >]\!] := [\![< className >]\!] - > allInstances()$$

In this example, the semantics of the APRIL *AllInstances* atomic formula is translated into its corresponding OCL operator. The concrete usage in a stepwise translation of filter operation materializing as APRIL *select* operator may look like:

1. ⟦`each customer in all instances of Customer where Name = 'Tom' `⟧

   In this step, the following translation is chosen:

   `<SelectFunction> ::= <source>->select(<iterator>|<body>)`

2. ⟦`all instances of Customer`⟧`->select(`⟦`customer`⟧`|`⟦`Name`⟧⟦`=`⟧⟦$'Tom'$⟧`)`

   The second step maps the parameters of the APRIL atomic formula to the parameters of the OCL operator. Whereas, the `<body>` part gets translated into an OCL expression using a relational infix operator with two parameters.

3. `Customer.allInstances()->select(`⟦`customer`⟧`|`⟦`Name`⟧⟦`=`⟧⟦$'Tom'$⟧`)`

   In this step, the AllInstances function gets translated. In this context, the meaning of the class name ($[\![< className >]\!_{CM}$) is defined as an element of the set of class names defined in the corresponding UML-class model (CM) (see Figure 4.1). The respective UML-class model spans the graph upon which expressions are specified. Thus, for referring single objects or sets of objects, class names are used

in combination with certain functions, or role names, attached to association ends. Each name of a UML class is a sequence of alphanumeric characters with special conventions of ordering described in the UML specification [84]. In this example, the parameter `<className>` is occupied by the value `Customer`.

4. `Customer.allInstances()->select(customer | Name = 'Tom')`
   At last, the low layer tokens get translated. Thus, `<iterator>` (here occupied by value `customer`) gets translated into `customer` as user-defined iterator variables are adopted unchanged. The property `Name` also gets adopted unchanged. However, note that if navigation paths are used, APRIL allows abbreviating them in some special cases (see Section 4.4.1), which has to be resolved in a preceding step. The basic operator = as well as basic types like integers and strings (here `'Tom'`) also get adopted as they are. Hence, the final translation step yields the OCL expression.

4.2.6 Extending APRIL with custom atomic formulas

Like definitions, customizable atomic formulae are verbalized as textual business patterns. Here, a requirements engineer can reuse his already existing, informal textual business patterns [98], which, unlike the more abstract APRIL definitions, express a very basic business rule- or business process pattern that occurs sufficiently often in the specification, justifying its incorporation into the language as an integral concept. Please note that adding specific atomic formulae into the language fundamentally binds the new APRIL derivative to the target domain, which makes the derivative less suitable to be used in different domains. The reason is that the more atomic formulas are in the grammar the higher the risk of producing ambiguities, making workarounds necessary to bring in new atomic formulae. Making workarounds of production rules in formal grammars can result in adapting the originally intended syntax, which could lead to a deviation from the originally and carefully developed natural language expression.

However, once a requirements engineer has developed an unambiguous pattern with the exact intended natural language reading he wants to incorporate into the grammar, e.g., for verbalizing a certain business behaviour, he/she can do so as the following example sketches.

The example, business process statement specifies that in a warehouse, all elements in a goods-stock move from a given target to a dedicated truck-loading bay and have to pass a certain to-be-defined gate on their way. Therefore, the resulting new atomic formula is added to the list of production rules in the grammar, respectively. Generally, a context-free grammar consists of a start symbol, production rules, terminals, and non-terminals [2]. Therefore, a state of practice language implementation mechanism described by Parr [91] is used. Formal production rules are used to generate text recognition algorithms of a parser that processes statements of a language to generate a parse tree. Second, a parse tree rewrite rule has to be specified along with the production rule. Parse tree rewrite rules are instructions for the parser on how to construct that part of the *abstract syntax tree (AST)* that reflects the recognized syntax of, for example, an atomic formula.

The abstract syntax tree is a condensed version of the parse tree that can be influenced by semantic considerations to form a concise and expressive logical representation of the parsed statements. For APRIL, the abstract syntax tree provides the necessary flexibility to incorporate user-defined language parts and also makes it particularly easy to extract the necessary parameters for the compiler. For clarification, Listing 4.4 sketches the definition of a user-defined atomic formula. It formalizes the example operator that reflects the scenario mentioned above. In Line 1, the production rule with the name of the non-terminal (atomic formula) *moveTo* is introduced. The definition of the new atomic formula's regular syntax is defined in the Lines 2-7. Here, the non-terminal *referenceOrDefinitionCall* is similar to that in Listing 4.3. This non-terminal is a predefined APRIL concept and can either refer to an element of the related domain model (e.g., to class names Store, Bay, Gate) or to values in the scope stack of the parent rule or definition in which the formula is used. The references to the parse tree nodes

of type *referenceOrDefinitionCall* in the Lines 3, 5 and 7 are stored one by one in the local variables source, target, and routeNode. Line 9 concludes the specification of the grammar rule with the parse tree rewrite rule. It is delimited from the syntax rule by the "→" sign. It tells the parser to construct a tree with the MOVETO terminal as root node having three leaves: source, target, and routeNode.

```
1  moveTo ::=
2  'all elements in'
3  source=referenceOrDefinitionCall
4  'move to'
5  target=referenceOrDefinitionCall
6  'over'
7  routeNode=referenceOrDefinitionCall
8
9  → ^(MOVETO $source $target $routeNode);
```

Listing 4.4: Grammar rule and parse tree rewrite rule for the operator **moveTo** in ANTLR 3.0.

The grammar rule and the parse tree rewrite rule in Listing 4.4 get injected into dedicated areas of the APRIL core grammar. The parameterization of the APRIL compiler is straightforward, which is depicted in Figure 4.2. In the second pass, a so-called

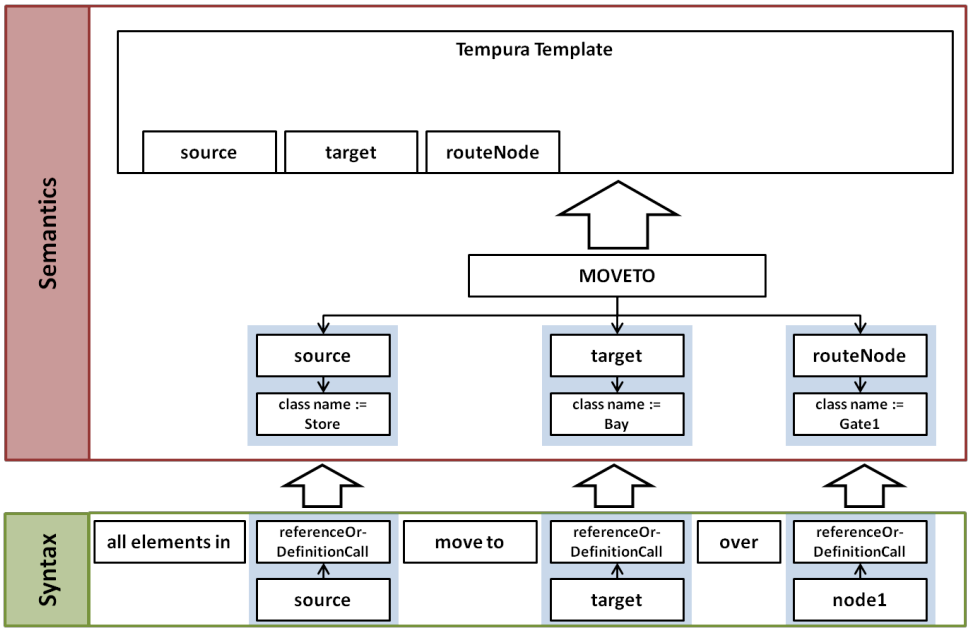


Fig. 4.2: Translation example of the atomic operator **moveTo**.

tree parser interprets the AST (of the rewrite rule MOVETO) and decides, which target language template to apply to the AST of the atomic formula. It then passes the values of the leaf nodes (here the values of the variables $source, $target, and $routeNode) to the parameters of the respective template (see Figure 4.2). The instantiated template is the actual translation of the atomic formula into the target language, representing the semantics of the respective operator. Please see Listing 4.5 as an example instantiation. With regard to the model, the Tempura statements in Listing 4.5 hold. They are actually an instantiation of a template that is used by the APRIL-compiler for translating the move-to-operator if used in an APRIL statement like in Listing 4.6. The formatting of the statements is according to String Template described by Parr [93] and contains generic parts that get filled according to the parameters of the operator in Listing 4.6.

```
define store_moves_to_Bay_over_Gate1 () = {
   len(|OUTPUT|-1) and
   I = 0 and
   I gets I+1 and
   moveAtoB(OUTPUT[I][Store], OUTPUT[I][Gate1]) and
   moveAtoB(OUTPUT[I][Gate1], OUTPUT[I][Bay]) and
   OUTPUT[|OUTPUT|-1] [Bay] ← OUTPUT[0] [Store]
}.

define moveAtoB (A,B) = {
   if (|A| > 0) then {
      first(A) gets last(B) and skip
   }
}.
```

Listing 4.5: Template for the all-elements-move-to operator.

all elements in Store **move to** Bay **over** Gate1.

Listing 4.6: Usage of the all-elements-move-to operator.

4.2.7 Frequently used business rules as atomic formulae

An important aspect that increases the expressiveness of APRIL is the utilization of language constructs that allow to briefly specify business rules. These abbreviations

are frequently used in practice and would be somewhat complicated to formulate in the underlying target languages. Such constraints are also often referred to as common constraints [43]. Costal et.al. [20] show that these types of business rules can cover a significant amount of the overall constraints occurring in real life systems. Section 4.5 presents the relevant common constraints as default atomic formulas integrated into APRIL.

4.3 The Universe of discourse

APRIL defines constraints on UML-Class models, which provide the names of the concepts to be used in a constraint. A meaningful naming helps to comply with an appropriate standard of natural language reading in the APRIL constraints. Most of the following style guide rules are anyhow proposed by several authors, e.g., [10]. However, it is reasonable to at least abide by the following style rules, which concern natural language grammar rules (nl) and/or rules that can be formally defined on a meta model (mm) level, making them easier enforceable automatically.

Rules for Classes:

- Every class name has to be unique. (mm)
- A class name begins with an uppercase letter. (mm)
- A class name is a singular substantive. (nl)

Rules for Attributes:

- Attributes are in singular form if they do not specify set; if they do, they have to be in plural form. (mm, nl)
- Attributes of Classes may not be typed by a class type of the domain model (implementation technique). This must be solved via an association. (mm)
- (optional) Attribute names are substantives (nl)

Rules for Associations:

- Every association end has to have a role name. (mm)
- Every association end has to have a multiplicity statement that allows to reason if its cardinality is one or many. (mm, nl)
    - Role names at association ends with cardinality "one" are in singular form. (mm, nl)

    - Role names at association ends with cardinality "many" are in plural form. (mm, nl)

- Role names have to be in lower case. (mm)
- Role names have to be substantives. (mm)
- Role names should be unique. (mm)

A rule of thumb, which can widely be found in literature, is that the description of domain concepts shall require as few model elements as possible and as many as necessary. Some additional model guidelines for creating domain models in UML are given by van Lamsweerde in [119]. Here are some of these:

- *Usage of association classes*: If some association between two classes needs additional characterization, especially if the characteristics change over time, it is easier to track the changes.
- *Avoid non-structural links*: If an attribute is of a type of another class in the same domain model, consider replacing it by an association.
- *Avoid obscure concept names*: The naming of any named entity shall be as abstract as necessary but as specific as possible.
- *Do not mix system views*: Keep the domain models clean of technical aspects. Only model the domain knowledge and leave implementation details out.

4.4 Rule syntax abbreviation Norma

In the following subsections, a list of syntax abbreviation methods is shown that can make certain APRIL statements more natural. The first one is the automated path deduction that can inference a chain of collect operators from a single statement. A very trivial means to make a statement more readable is to replace it with a symbol, which often materializes in a decomposition of a more complex statement. This holds for almost any language but for the sake of completeness, it is contained here as a subsection. The simplification of iterator statements, on the other hand, is an abbreviation mechanism that is tailored to APRIL's operators dealing with sets. The section is concluded with a presentation of method groups, allowing a shorthand for function calls omitting the parameters if they are deducible from the context the function call is used in.

4.4.1 Automated path deduction

The idea behind automatic path deduction is to substitute the technical expression in a "dot"-notation with a reduced statement of an improved natural language reading. If the starting node with the context class and target nodes with the respective role names are given, it is possible to automatically deduce a path from a single role statement. This can only be done if there is no other path in the starting and the target node that is equal. The following expression contains a long navigation – over two hops – which is pretty awkward with regard to a natural-language sentence. E.g., a business rule based on Figure 4.3 stating that: *"A student may only attend subjects of his course"* can look like:

```
1 Invariant Inv1 concerns Student:
2 lectures.attendedSubject is in course.offeredCourseSubjects .
```

The reduced statement with a deduced path is:

```
1 Invariant Inv1 concerns Student:
2 attendedSubject is in offeredCourseSubjects .
```

In examining the example model, the navigation from *Student* to class *Subject* – stated by the role name *attendedSubjects* – can be achieved by navigating along *lectures* to *attendedSubject* and *grades* to *exam* to *lecture* and finally to *attendedSubject*, which means that, in this case, there is no unique navigation path. Thus, the first statement cannot be resolved in this way. NOTE: The deduction rule neglects cyclic navigation paths. This means that any role serving as the target node of a navigation path is unique in that navigation path. E.g., a navigation starting at class student looks like "lectures.attendedSubjects.owner.attendees.lectures." Any further cycle is not considered for resolving that path. Looking at the second argument of the "is in" operation, it can be seen that "offeredCourseSubjects" can only be reached along course.offeredCourseSubjects. Thus, the abbreviated form "offeredCourseSubjects" is valid in this context.

4.4.2 Replacement by decomposition

This replacement method is the manual alternative of the automatic path deduction (see Section 4.4.1). In order to replace a long navigation statement (e.g., over more than two navigation hops), an APRIL definition can be used (see also Section 4.2.3). The benefit of decomposing generally applies to any long and awkward-sounding statement that can be replaced by a single pithy word representing, e.g., an APRIL definition or local variable.

4.4.3 Simplification of iterator statements

APRIL provides means to optionally make the syntax of atomic formulas with iterators more concise. For example, a statement in a rule based on the model in Figure

4.3, having the context *Lecture*, which states that "`every student in students satisfies that <exp>`" can be reduced to the simpler form stating, "`every student satisfies that <exp>`". Therefore, APRIL uses the following rules:

1. Given a role-naming convention which states that association ends shall only be named with nouns, and association ends with cardinality "many" shall be named in the plural form, the syntax-based inference rule is as follows:

   A function, e.g., `every student satisfies that <exp(student)>` shall be expanded to `every student in students satisfies that <exp(student)>`.

   - Converting the name of the iterator variable into the plural form by usually adding a plural "-s" in natural language, for the sake of simplicity, is adopted as a universal rule in APRIL: *plural(student)* = *students*. Exceptions to that rule imposed by natural language could be handled with a singular-to-plural mapping table for a better natural language reading.
   - Try to resolve the superset with `plural(student)` by utilizing the navigation abbreviation rule mentioned earlier.

2. If this algorithm is used in a definition, the parameter names are used for resolution before the role names. Generally, an APRIL definition inherits its context from the rule it is used in.

4.4.4 Method groups

In the programming language C#, the term method group is used to denote one method out of a set of methods invoked, which have equal constant signature parts and only differ in the typing of their parameters. The benefit of using a method group as a call is to omit the parameter as the statement in which it is used in allows inferencing the parameter. By omitting the parameter, the entire expression gets an additional degree of freedom

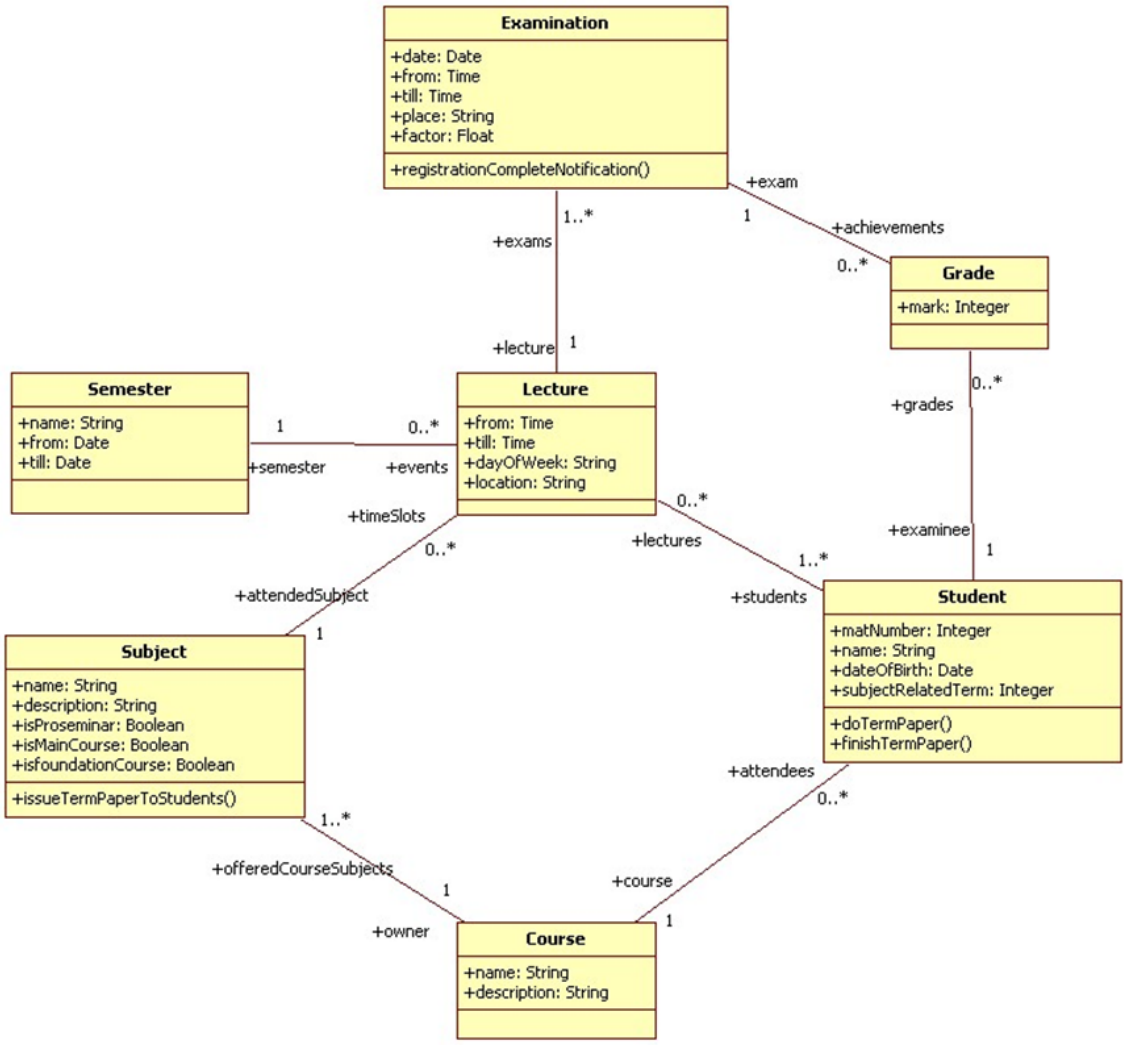


Fig. 4.3: University organisation

towards its natural language design as one less constituent may have to comply with linguistic grammar requirements. Generally, a method group can be inferenced if the typing of a local method definition can be mapped one to one to the local context. This means that the current scope stack contains one definition matching the constant part, along with the properly typed parameter appendix. In APRIL, method group resolution is only applicable to local method definitions. Please consider Listing 4.7 which shows an example statement using two nested atomic formulas in which the embedded formula uses the iterator variable *customer* to parameterize the local method *"ordered products"*.

As the embedded statement uses the locally defined method *"ordered products"* with *orderer* of type *Customer* as the only parameter in the scope of the embedding statement

```
...
is defined as
   every customer in customers satisfies that
      every "ordered product" in "ordered products"(customer) satisfies that
...
   with
      "ordered products" (orderer as Customer) is defined as
...
```

Listing 4.7: Method group example: fully expanded statements with no method group resolution

```
...
is defined as
   every customer in customers satisfies that
      every "ordered product" in "ordered products" satisfies that
...
   with
      "ordered products" (orderer as Customer) is defined as
...
```

Listing 4.8: Method group example: local method call with method group resolution

introducing an iterator variable of the same type, the mechanism to resolve the appropriate method group is able to unambiguously map to the symbols *"ordered products"* of the call to the local method.

Hence, the statement in Listing 4.7 can be abbreviated to the method group form of Listing 4.8.

4.5 Common constraints

A central aspect that increases the expressiveness of APRIL is the utilization of language constructs that allow specifying business rules briefly. These abbreviations are frequently used in practice and would be somewhat complicated to formulate in the underlying target languages. Such constraints are also often referred to as common constraints. Costal et.al. [20] show that these types of business rules can cover a significant amount of the overall constraints occurring in real life systems. In order to give the presented common constraints a structure, we have grouped them together based on the taxonomy presented in Figure 4.4, which was inspired by Halpin et al. [43] and Miliauskaite et al. [71, 72] and will be explained in the following subchapters. For the following examples of common constraints, please consider the domain model (UML-class model) in Figure 4.5.

For a better comprehensibility for the reader, the two UML class model concepts are explained:

- The ternary association diamond between the classes *Order*, *Shipment* and *Product* describes a tuple of related objects of these types. They are used for special identification constraints of the composed objects.

- The combination of a composition association combined with an inheritance between the classes Package and Product. This can be used to construct recursive object trees. Please note that the generic class (*Package*) inherits the attributes from its generalized counterpart (*Product*). As roles are modelled a attributes, they can be access from each of the classes accordingly.

4.5.1 Constraints on values

Restricting the values of variables can be done in several ways. For example, assigning an integer data type to a variable restricts its values to a given range of natural numbers. Another way is to use relational operators with, e.g., constants to explicitly constrain

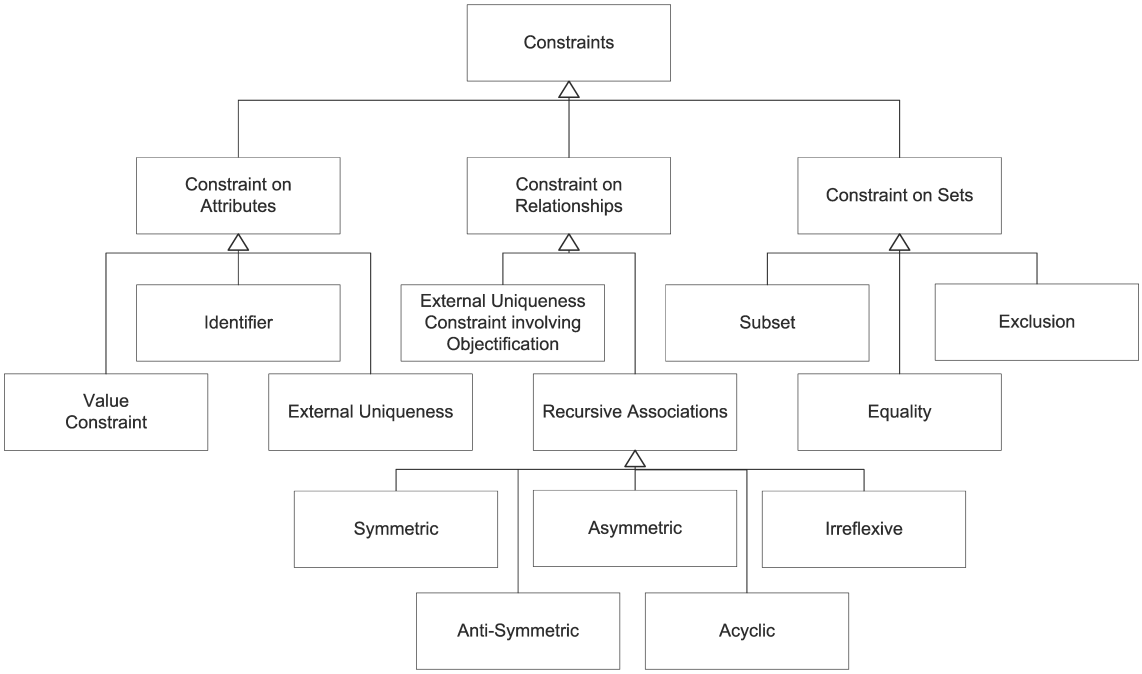


Fig. 4.4: Taxonomy of some important common constraints.

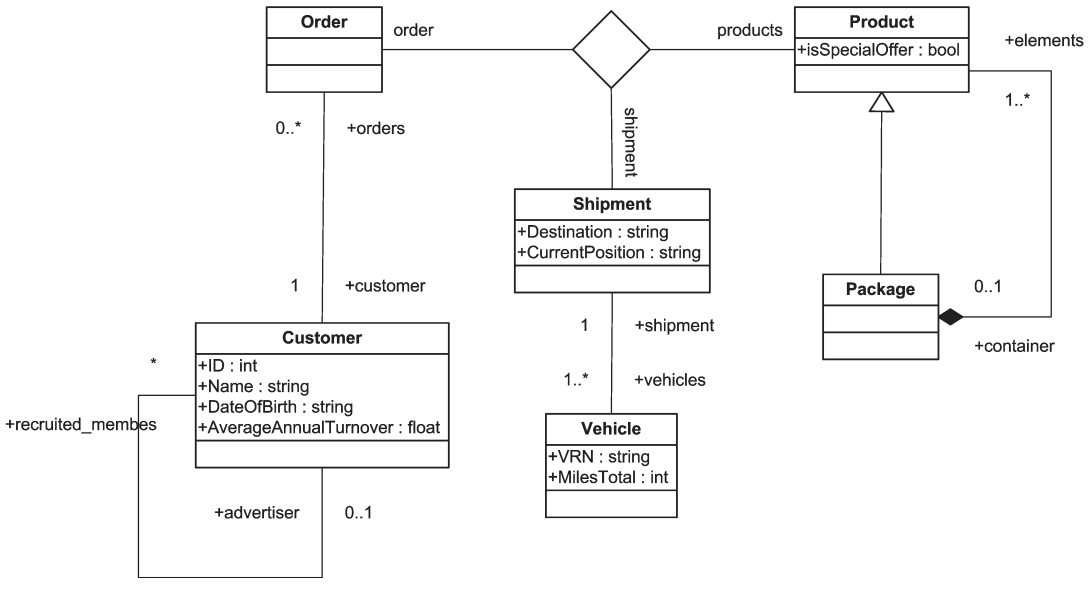


Fig. 4.5: Domain model for the running examples on common constraints

variables. Therefore, the conventional and well-known binary relational operators (e.g., $<, >, =, <>, <=, >=$) are used. Although APRIL's is meant to be close to natural language, a mathematical representation of the aforementioned operators is used as atomic formulae as they should be known to anyone. Moreover, if this might be too disconcerting for a user to use in a language like APRIL, it is possible to redefine that particular part of the grammar to give these operators a natural language syntax (e.g., "A>B" may become "A greater than B"). Here is an example:

APRIL:

```
1 Invariant Values concerns Vehicle:
2 MilesTotal < 100000 .
```

4.5.2 Identifier

According to Miliauskaite et al. [72], a useful and highly demanded constraint is the identifier or primary identifier known to ERM [79], ORM [42], xUML [67], and relational database management systems (RDBMS). UML's class attributes are predestined for holding a primary identification rule stated in APRIL or OCL, as UML class diagrams by default lack such means. This can be shown with the help of the model in Figure 4.1. A common scenario is that an object is identified by an attribute that carries a unique value over all objects of the entire population. This can be formalized as follows:

APRIL:

```
1 Invariant <RuleName> concerns <ClassName>:
2 <AttributeNameOfClass> is unique.
```

OCL:

```
context <ClassName> inv <RuleName>:
Service.allInstances->isUnique(serviceNr)
```

A more general version of the primary identifier constraint is the internal uniqueness constraint, also called composed identifier. It states that value combinations of two or more attributes of an object are unique [43] [72]. The APRIL and OCL versions look like:

APRIL:

```
Invariant composedId concerns Customer:
each Name, DateOfBirth combination is unique .
```

OCL:

```
context Customer inv composedId:
Customer.allInstances->forAll(c1,c2 | c1 <> c2 implies
not((c1.Name = c2.Name and c1.DateOfBirth = c2.DateOfBirth)))
```

Towards a reasoning of common constraints, it is valid to say that if a class has a tuple of attributes $a_1, ..., a_i$ out of which at least one has to obey a primary identifier constraint, it is redundant to additionally specify that the combination of $a_1, ..., a_i, ..., a_n$ has to obey a composed identifier constraint.

4.5.3 External uniqueness

A uniqueness constraint is denoted as external if the identification scheme is bound to another attribute of an associated class. For example, an object *a* may be related to an object *b* only once. This does not restrict the overall occurrence of object *b* throughout

the entire model as it might be that *a* is also related to an object *c*.

In the example below, the constraint only holds if different instances of `Vehicle` with potentially equivalent values in `VRN` (vehicle registration number) are not linked to the same instance of `Shipment`. The combination of `Shipment` and the attribute `VRN` of the class `Vehicle` is inherently done by the navigation path from `Shipment` to `Vehicle` by stating its concrete role name, `vehicles`. This collects all instances of `Vehicle` that are linked with the current instance of `Shipment`.

APRIL:

```
1 Invariant externalUniqueness concerns Shipment:
2 VRN is unique in vehicles.
```

OCL:

```
1 context Shipment inv externalUniqueness:
2 vehicles->isUnique(VRN)
```

4.5.4 External uniqueness involving objectification

This type of constraints deals with associations that are regarded as objects. Hence, objectification [41], known as reification in UML, aims to combine multiple classes or attributes to a single one in order to apply constraints on the combination. In UML, this is typically done using association classes which objectify the association between two classes. Hence, an object of an association class identifies a unique n-tuple of linked objects. In an attempt to generalize several UML concepts, Gogolla et al. [38] show how to transform association classes and association qualifiers into n-ary associations (with n as a natural number and $n \geq 2$). As they show in [37], there are several problems with the use and constraining of n-ary associations and thus, the n-ary association has to be

transformed into a proper set of binary associations. For our example in Figure 4.1, it would mean that the association diamond in the middle of the three associated classes (Order, Shipment, Product) is transformed into an additional synthetic class, e.g., called ASSOC, being associated to each of the aforementioned classes with a binary association. In order to handle objectification in business rule statements between two or more Classes $c_1, ..., c_N$, the APRIL *"each $c_1, ..., c_N$ combination"* expression is used. It returns a set of synthetic association objects instantiated from class ASSOC, each of which is associated with one object of the corresponding type of $c_1, ..., c_N$. The first example shows how to constrain tuples of classes. The prose explanation for the APRIL constraint is omitted here because it should be self-explanatory.

APRIL:

```
Invariant externalUniqueness concerns Product:
each Product, Order, Shipment combination is unique.
```

OCL:

```
context Product
inv externalUniqueness:
   Product.allInstances->forAll( c |
      Order.allInstances->forAll ( b |
        Shipment.allInstances->forAll ( a |
         ASSOC.allInstances->select( assoc | assoc.product = c
   and assoc.order = b and assoc.shipment = a )->size()<=1 ) ) )
```

4.5.5 Recursive associations

A UML class can be associated with itself (see class Product in Figure 4.1). This allows recursions between objects. The rules on such models are called ring constraints [43].

Common ring constraints follow typical association properties. Here are some examples:

4.5.5.1 Irreflexive

Irreflexive constraints do not allow objects to refer back to themselves, which is formally stated below. Note that the OCL keyword `self` corresponds to the lowercase class name in the APRIL rule body.

APRIL:

```
Invariant irreflexive concerns Package:
package is not in elements.
```

OCL:

```
context Package inv:
elements->excludes(self)
```

4.5.5.2 Transitive

This type of constraint states that if a first object bears the relationship to a second and the second to a third one, then the first also bears a relationship to the third. This can be formally stated as follows. The example for a transitive business rule is demonstrated based on a recursive function that introduces a transitive closure. In our example, this means that if a customer recruited other customers and each of them also recruited customers, the original recruiting customer is related to all of the subordinate customers along the link derived from the "*advertiser-to-recruited_members*"-association. In APRIL, the deep collection operator traverses the related objects having a self association.

APRIL:

```
Invariant transitive concerns Customer:
number of recruited_customers > 10
with
recruited_customerns is defined as deep collection of
   recruited_members .
```

OCL:

```
context Customer inv transitive:
let recruited_customerns : Set(Customer) = self.closure() in
recruited_customerns.count() > 10


context Customer def closure() : Set(Customer) =
 recruited_members.closure()->asSet()->including(self)
```

4.5.5.3 Intransitive

If a first object bears the relationship to a second and the second to a third one, then the first cannot bear a relationship to the third.

APRIL:

```
Invariant intransitive concerns Customer:
recruited_members.recruited_members is not in
 recruited_members.
```

OCL:

```
context Customer inv intransitive:
recruited_members.recruited_members->
  excludesAll(recruited_members)
```

4.5.5.4 Symmetric

If the first object bears the relationship to the second, then the second bears the relationship to the first. However, a ring constraint defined in a UML class diagram is by default symmetric so there is no need to state that an association is symmetric if it is meant to be optional. If not, the following example shows how it can be stated.

APRIL:

```
Invariant symmetric concerns Customer :
this is in recruited_members.recruited_members .
```

OCL:

```
context Customer inv symmetric:
recruited_members.recruited_members->includes(self)
```

4.5.5.5 Asymmetric

If the first object bears the relationship to the second object, then the second cannot bear the relationship to the first.

APRIL:

```
Invariant asymmetric concerns Customer :
this is not in recruited_members.recruited_members .
```

OCL:

```
context Customer inv asymmetric:
recruited_members.recruited_members->excludes(self)
```

4.5.5.6 Anti-Symmetric

If the objects are different, then if the first bears the relationship to the second, then the second cannot bear that relationship to the first.

APRIL:

```
Invariant antiSymmetric concerns Customer :
this is not in recruitedMembers.recruited_members
with
recruitedMembers is defined as
each customer in recruited_members where customer <> this .
```

OCL:

```
context Customer inv:
recruited_members->select(a | a<>self).recruited_members->
   excludes(self)
```

4.5.5.7 Acyclic

A chain of one or more instances which are linked to objects of the same type cannot form a cycle (recursive loop). As acyclic constraints are common practice in business rule modelling [43], and their definition is quite tricky, a new language construct for APRIL is justified, which is incorporated as an atomic formula. That is the APRIL *"deep collection of"* - operator, which is based on the OCL-version of Costal et al. [20] presented later. The semantics of the operator is as follows:

Let A be a set of objects of type τ and $\{a_0 \dots a_n\} \cup \emptyset$ be elements of A. Let $R_n(a_n$, $A_n := \{a_{n+1,j} \dots a_{n+1,k}\} \backslash \emptyset$) be the representation of a set of relations between an element a_n and a non-empty set $A_n \subset A$.

Then $A_{deep} := \bigcup A_m(R_m)$; with $0 \leq m \leq n$. After all $(a_0, A_{deep}) \models$ *"deep collection of"*(a_0).

APRIL:

```
Invariant acyclic concerns Package:
package is not in deep collection of elements.
```

OCL:

```
context Package def:
successors(): Collection(Product) =
self.elements->
 union(self.elements.successors())

context Package inv acyclic:
self.successors()->excludes(self)
```

In the OCL example, the first constraint defines a synthetic operation named

`successor()` that is recursively called within an OCL union operation which is called on the set of `elements` of the current object. The intent is to unite all the `Package` objects linked with the current objects `elements` that also play this role in the linked subordinated objects. This construct is inherently typed as collection type `Collection`.

4.5.6 Sets

The upper part of Table 4.1 shows the verbalization of common constraints on sets according to Costal et al. [20] and Miliauskaite et al. [72]. Note that lowercase letters denote elements of sets and uppercase letters denote sets.

The lower, folded part of the table handles a specialization of the natural join operator, indicated by $\bowtie'$. In APRIL, this is used to "navigate" through UML class models, gathering (sets of) objects along association graphs, which can then be utilized to formulate constraints. Hence, this is one of the most important constructs in APRIL and deserves special attention. The semantics is equivalent to OCL's [85] *collect* operation.

APRIL	OCL	mathematical
a *is in* A B *is in* A	$A \rightarrow$ *includes* (a) $A \rightarrow$ *includesAll* (B)	$a \in A$ $B \subseteq A$
$A = B$	$A = B$	$A = B$
A *is not in* B a *is not in* A	$B \rightarrow$ *excludesAll*(A) $A \rightarrow$ *excludes*(a)	$A \cap B = \emptyset$ $a \notin A$
$A.B$	$A.B$ $A \rightarrow$collect(B)	$A \bowtie' B$

Table 4.1: Some common constraints on sets.

4.5.6.1 Subset

This constraint denotes that a role population between two or more object types (e.g., UML classes) are in a set-subset relation. Consider Table 4.1, which shows the subset operators. Please note that in APRIL, equal syntax is used for checking if a single object or a set is a sub-element or -set. The compiler has to inference if the left side of the set

operator is a set or single object, in order to select the correct transformation template of the target language operator (see the *"is in"* operator in Table 4.1).

4.5.6.2 Equality

Checking collections for equality in APRIL as well as OCL is simply done with the equation operator (=) that is also applied to collections as well as primitive types like string, integer, etc.

4.5.6.3 Exclusion

The typical scenario for exclusion constraints is when two sets are supposed to be mutually disjoint or their intersection constitutes the empty set (see the *"is not in"* operator in Table 4.1). Again, the compiler has to decide which translation template to select based on the type of the left constituent of the operator.

4.6 The rationale of APRIL

This section presents the syntax and semantics of APRIL.

4.6.1 Syntax

In this section, the definition of the syntax of the APRIL core part of the language is given. Therefore, the Extended Backus-Naur Form (EBNF) is chosen, since it is typical for defining formal grammars. Moreover, EBNF is understood by the utilized compiler generator framework ANTLR 3.0 (ANother Tool for Language Recognition (ANTLR)). The subsections of this section are organized as follows. The introduction Section 4.6.2 contains some of the basics of EBNF. For a deeper insight into EBNF, the interested reader should consult the relevant literature. The Section 4.6.3 contains a top-down EBNF-based definition of the basic building blocks of an APRIL rule or definition. The basic operators and their precedence is presented in Section 4.6.4. The function lexicon in Section 4.6.5 contains the default atomic formulas of APRIL. The chapter is concluded with the presentation of the elementary constructs, including the start symbol and some essential terminals in Section 4.6.6.

4.6.2 Brief introduction to APRIL's formal grammar definition

Basically, a formal grammar requires a start symbol, production rules, terminals, and non-terminals [2]. In the context of the APRIL grammar, the start symbol is defined in Line 100 of Listing 4.6.4 as well as the most important terminals in the Lines 101, 103, and 105. The rest of the terminals occur on the right side of the production rules for structuring statement definitions. Production rules are of the form <*left*> *::=* <*right*>*;*. This means that nonterminal symbols are the *left* side of the production rule since they get replaced by the *right* side, which is either other nonterminals or groups of terminal symbols or a mixture of both. Constructing a production rule requires defining a symbol for the production rule that is denoted by the left side of an EBNF statement. Replacing the left

side consisting of a single nonterminal by the right side is a characteristic of context-free grammars. A very simple example of a production rule that shows the substitution of a nonterminal <*Boolean*> of either the terminal *"true"* or *"false"* is

```
1 <Boolean> ::= "true" | "false" ;
```

The utilized EBNF syntax uses angle brackets <Boolean> to enclose the name of the nonterminal symbol, followed by the "::=", which means that the nonterminal (left side) is substituted by the right side. Terminals are enclosed by quotation marks. The entire production rule is concluded by the semicolon character. EBNF also introduces control symbols denoting cardinalities of (non-)terminal symbols. The next example uses curly brackets to denote that the inner terminal alternation can occur zero to many times. Assigning cardinalities can also be done using the square brackets for the one-to-many cardinality. A different grouping mechanism is employed by the round brackets denoting that a group of (non)terminals belong together and must occur exactly once. The pipe character ("|") is used to indicate an alternation that states that exactly one of the alternatives is supposed to match.

```
1 <ID>::= ("a" .. "z" | "A"..."Z" | "_" ) {"a" .. "z" | "A"..."Z" | "_" | "0" .. "9" };
```

Terminals and nonterminals can be mixed at the right side of the production rule to predefine the structure of a statement. For example, the production rule for the invariant requires an introductory terminal *"Invariant"* followed by a name that can be produced by the <*ID*> nonterminal and then by the *"concerns"* terminal and so on.

```
1 <Invariant> ::= "Invariant" <ID> "concerns" <ClassName> ":" <Rule> ;
```

For more detailed information on formal grammars and on EBNF, the interested reader should consult the relevant literature, e.g., by Aho et al. [2] or Parr's book on ANTLR

[91].

4.6.3 Rule and definition headers

The syntax of any APRIL definition or rule (as defined in Listing 4.6.1) consists of a header introduced with a starting keyword denoting its purpose that can either be an invariant, pre- or post-condition, or a rule of behaviour. Typically, the definition is selected for reusable statements. The header part contains a characteristic property to identify the rule or definition. It concludes with the colon character (":") either after the *concerns* <*Classname*> or the *yielding* <*Type*> passage (see Lines 3 to 13 of Listing 4.6.1). The body part follows the header part (Line 15 in Listing 4.6.1), in which the actual statement is specified. The conclusion of the body part is done by using either the *"."* or the *"with"* keyword both followed by a *newline* (Line 19 of Listing 4.6.1). Using *"with"* for terminating the body part introduces the local variable/method definition section at the same time. Here, local auxiliary statements can be specified for small-scale decomposition purposes. Again, the *"."* finally concludes the local definition section and with it, the entire rule or definition.

```
1
2   <Rules> ::=  <Invariant> | <PreOrPostCondition> | <Behaviour>;
3
4   <PreOrPostCondition> ::= ("Precondition" | "Postcondition") <ID> "concerns" <
    OperationName> ":" <Rule> ;
5
6   <Invariant> ::= "Invariant" <ID> "concerns" <ClassName> ":" <Rule> ;
7
8   <Behaviour> ::= "Behaviour" <ID> "concerns" <ClassName> ":" <BehaviouralRule> ;
9
10  <Definitions> ::= <FilterDefinition> | <ExtensionDefinition> | <ValueDefinition> ;
11
12  <ValueDefinition> ::= ("Definition" | "Value" | "Filter") <MixFixName> ["yielding" <
    typeDefinition>] "is defined as" <Rule> ;
13
14  <Rule> ::= <LogicalExpression> <RuleEnd> ;
15
16  <RuleEnd> ::= ("." | <LocalVariableDefinitionBlock>) ;
17
18  <BehaviouralRule> ::= <TempuraExpression> ("." | "with" <
    TempuraLocalVariableDefinition> {"," <TempuraLocalVariableDefinition>} ".") ;
19
20  <TempuraLocalVariableDefinition> ::= <ID> "is defined as" <TempuraExpression>;
21
22  <LocalVariableDefinitionBlock> ::= "with" <LocalVariableDefinition> {"," <
    LocalVariableDefinition>} "." ;
23
24  <LocalVariableDefinition> ::= (<ID> | '' <ID> { <ID>} '') "is defined as" <
    LogicalExpression> ;
```

Listing 4.6.1: Rule and definition headers

4.6.4 Logical, relational, and arithmetic constructs

This section treats the general logical, relational, and arithmetic logical, relational, and arithmetic- (LRA-) functions, beginning with the <*LogicalExpression*> nonterminal in Line 29 of Listing 4.6.2. A logical expression is the link point between the <*Rule*>-body statements and the possible hierarchical combinations of the general LRA functions. The logical expression only resolves operators that yield a truth value because, in APRIL, rules are defined as predicates yielding truth values. The definition of the grammar implicitly supports the typing for the general LRA functions and hence, makes subsequent type-checking mechanisms obsolete, which supports the overall performance of a parser. However, the order of the production rule of Listing 4.6.2 incorporates a hierarchy obligating precedence rules to the grammar, which makes it possible to use the general operators in the usual way. So, for example, a statement of the form ***1 + 2 < 5 and (-12)*2 <> 99 or not(true)*** does not have to be grouped explicitly. With the precedence rules, grouping is implicitly defined by the structure of the production rules. Here is the correct grouping defined by the grammar, which is compliant to how the reader would expect it: ***((1 + 2) < 5) and (((-12)*2) <> 99) or (not(true))***. At the topmost level, the Boolean operators (*and* and *or*) force the overall expression to be of type Boolean. One level below, relational operators (< and <>) also yield a truth value. Relational operators typically allow predicating over element types in ordered sets such as the set of natural numbers. This grouping of production rules continues down to the <*Value*> nonterminal, which can branch out to any kind of typed concept usable in arbitrarily defined functions (see <*Function*> nonterminal). In Line 46 of Listing 4.6.2, the grammar allows recursively producing explicitly grouped LRA functions using round brackets. Whereas in Line 48, the link point to the atomic formulas is introduced by the nonterminal <*Functions*>.

```
<LogicalExpression> ::= <AndExpression> { ("or" | "implies that") <LogicalExpression>} ;
<AndExpression> ::= <RelationalExpression> {"and" <AndExpression>} ;

<RelationalExpression> ::=
<AdditionExpression> {(">" | "<" | ">=" | "<=" | "=" | "<>") <RelationalExpression>} ;

<AdditionExpression> ::=
<MultiplicationExpression> {("+" | "-") <AdditionExpression>} ;

<MultiplicationExpression> ::=
<NagationExpression> {("*" | "/") <MultiplicationExpression>} ;

<NegationExpression> ::= ["-" | "not" | "it is not the case that"] <Value> ;

<Value> ::= <Boolean> |
<ID> |
<Number> |
"(" <LogicalExpression> ")" |
<PathNameOrDefinitionCall> |
<Functions> ;

<Functions> ::= <SetFunctions> | <BoolFunctions> | <ValueFunctions> ;
```

Listing 4.6.2: Logical, relational, and arithmetic constructs

4.6.5 Function lexicon

In Listing 4.6.3, the production rules for APRIL's atomic formulas are specified. The hierarchy of the production rules distinguishes between functions that yield different type groups for implicit type resolution that allows omitting a separate resolution step once the parse tree is completely built by the parser. The implicit typing actually applies only to functions typed as Boolean. For set functions and value functions, the types have to be resolved explicitly. However, in the earlier sections, it is mentioned that APRIL is extensible by defining and adding atomic formulas to APRIL's grammar. If a new atomic formula is supposed to be added, the respective production rule has to be appended under one of the production rules *BooleanFunction*, *ValueFunction*, or *SetFunction*. By adding new production rules for atomic formulas, it is necessary that the grammar remains free of ambiguities since it is necessary to interpret a statement that pretends to comply with the grammar in a unique way.

```
1  <BoolFunctions> := <ForAll> | <Exists> | <IsEmpty> | <IsNotEmpty> | <Unique> | <
   Includes> | <Excludes> | <Equivalence> | <IsTypeOf> ;
2
3  <ForAll> ::= "every" <ID> [, <ID>] ["in" <PathNameOrDefinitionCall>] "satisfies that
   " <BooleanExpression> ;
4
5  <Exists> ::= "at least one" <ID> ["in" <PathNameOrDefinitionCall>] "satisfies that"
   <BooleanExpression> ;
6
7  <IsEmpty> ::= <PathNameOrDefinitionCall> "is empty" ;
8  <IsNotEmpty> ::= <PathNameOrDefinitionCall>  "is not empty" ;
9
10 <Unique> ::= <PathNameOrDefinitionCall> "is unique" |
11 <PathNameOrDefinitionCall> "is unique in" <Value> ;
12
13 <Includes> ::= <PathNameOrDefinitionCall> "is in" <Value> ;
14 <Excludes> ::= <PathNameOrDefinitionCall>  "is not in" <Value> ;
15
16 <Equivalence> ::= "for every" [<ID> "in"] <PathNameOrDefinitionCall>
17 "all or none of the following holds :" <BooleanExpression> {","  <BooleanExpression
   >} ;
```

```
18
19  <IsTypeOf> ::= <PathNameOrDefinitionCall> "is type of" <ClassName>;
20
21  <ValueFunctions> ::= <CountFunction> | <SumFunction> | <AtFunction> | <LastFunction>
     | <IfThenElse> ;
22
23  <CountFunction> ::= "number of" <ID> "in" <PathNameOrDefinitionCall> ;
24  <SumFunction> ::= "sum of" <ID> "from" <PathNameOrDefinitionCall> ;
25
26  <AtFunction> ::= "element at position" <Integer> "in" (<PathNameOrDefinitionCall> |
    <SetFunctions> ) ;
27
28  <LastFunction> ::= "element at last position in" (<PathNameOrDefinitionCall> | <
    SetFunctions> ) ;
29
30  <IfThenElse> ::= "if" <BoolFunctions> "then" <Functions> "otherwise" <Functions> ;
31
32  <SetFunctions> ::= <SelectFunction> | <AllInstancesFunction> | <
    ReachableObjectsFunction> | <UnionFunction> |  <IntersectionFunction> | <
    EachCombinationFunction> | <WithoutFunction> | <WithFunction> | <
    CollectionConversionFunctions> | <OfTypeFunction> ;
33
34  <SelectFunction> ::= "each" <ID> ["in" <PathNameOrDefinitionCall>] "where" <
    BooleanExpression> ;
35  <AllInstancesFunction> ::= "all instances of" <ID> ;
36  <ReachableObjectsFunction> ::= ("reachable objects along" | "deep collection of") <
    ID> ;
37
38  <UnionFunction> ::= "union of" <PathNameOrDefinitionCall> "with" ( <SetFunctions> |
    <PathNameOrDefinitionCall> ) ;
39
40  <IntersectionOf> ::= "intersection of" <PathNameOrDefinitionCall> "with" ( <
    SetFunctions> | <PathNameOrDefinitionCall> ) ;
41
42  <EachCombinationFunction> ::= "each" <AttributeOrClassNameList>  "combination" ;
43
44  <WithoutFunction> ::= <PathNameOrDefinitionCall> ("without" | "excluding") ( <
    PathNameOrDefinitionCall> | <SetFunctions> ) ;
45  <WithFunction> ::= <PathNameOrDefinitionCall> "including" ( <
    PathNameOrDefinitionCall> | <SetFunctions> ) ;
46
47  <OfTypeFunction> ::= <PathNameOrDefinitionCall> "of type" <ClassName>;
48
```

```
49    <CollectionConversionFunctions> ::= <Set> | <Bag> | <Sequence> | <OrderedSet> ;
50
51    <Set> ::= "collection" <PathNameOrDefinitionCall> "as set" ;
52    <Bag> ::= "collection" <PathNameOrDefinitionCall> "as bag" ;
53    <Sequence> ::= "collection" <PathNameOrDefinitionCall> "as sequence" ;
54    <OrderedSet> ::= "collection" <PathNameOrDefinitionCall> "as ordered set" ;
```

Listing 4.6.3: Function Lexicon

4.6.6 Elementary constructs

In Listing 4.6.4, the elementary constructs are specified, which are, next to others, those for addressing domain model concepts (*ClassName*, *OperationName*, or *NavigationPath*) or the *MixFixName*, used for definition signatures.

```
55      "StartSymbol" = <Model> ;
56      <ID>::= ("a".."z" | "A" .. "Z" | "_") {"a".."z" | "A" .. "Z" | "_" | "0".."9" };
57      <Number> ::= <Integer> | <Float> ;
58      <Integer> ::= ("0".."9") {"0".."9"} ;
59      <Float> ::= <Integer>.<Integer> ;
60      <Boolean> ::= "true" | "false" ;
61      <MixFixName> ::= (<ID> | <MNParamDefinition>) [ " " <MixFixName>] ;
62      <MNParamDefinition>::= "(" <ID> "as" <ID> ")" ;
63      <OperationName> ::= <ID> "(" [<ID> ["as" <ID>] {"," <ID> ["as" <ID>]}] ")" ;
64      <ClassName> ::= <ID> ;
65      <PathNameOrDefinionCall> ::= <Self> | [<preExpression>] <NavigationPath> | <
    MixFixName> ;
66      <Self> ::= "this" | "this" <ClassName>;
67      <NavigationPath> ::= <ID> {"." <ID>} ;
68      <AttributeOrClassNameList> ::= <NavigationPath> {"," <NavigationPath>} ;
69      <preExpression> ::= "former value of" <ID> ;
70      <typeDefinition> ::= <ClassName> | <BasicType> | <SetDeclarator> "of" <
    typeDefinition> | (<typeDefinition>);
71      <BasicType> ::= "Number" | "Boolean" | "String";
72      <SetDeclarator> ::= "Collection" | "Set" | "Bag" | "Sequence" | "Ordered Set";
```

Listing 4.6.4: Elementary constructs

4.6.7 Semantics

The semantics of APRIL is based on Tempura for rules defining behaviour and OCL for invariants. A denotational semantics for mapping APRIL expressions into OCL and Tempura-expressions is chosen. APRIL Rules that are introduced with the keyword *Behaviour* (see the grammar in Section 4.6.3) get translated into Tempura, whereas any other rule is translated into OCL. Any mapping of an APRIL operator or function f to its corresponding OCL or Tempura operator r in the tables below is a shorthand for $\mathcal{M}[\![f]\!] \rightarrow r$. ($\mathcal{M}$ stands for "meaning"). Later a shorthand for $\mathcal{M}[\![f]\!] \rightarrow r$ is used that looks like $[\![f]\!] \models r$, so both $\mathcal{M}[\![f]\!]$ and $[\![f]\!]$ denote the *meaning* of f.

An APRIL, an expression for non-behavioural business rules denotes a set of character sequences that have to be built obeying the <*Invariant*> or <*PreOrPostCondition*> production rule according to the syntax specification (see Section 4.6.1). Any character sequence of this sort can be translated to a character sequence of OCL, which obeys the production rules of OCL's syntax [85]. The semantic foundation of OCL is realized by several tool manufacturers according to the OCL specification. Moreover, the scientific community has undertaken a lot of effort to provide semantic underpinnings for OCL [11, 85, 115]. The presented semantics is, therefore, based on these results. Even if that work might not be finished or discussed, due to the existing implementations of OCL tools, OCL can still be used as a basis for the semantics. Moreover, the use of a seldom discussed OCL semantics is better than using natural language for describing business rules.

APRIL expressions that obey the <*Behaviour*> production rule 4.6.1 can be translated to character sequences according to the Tempura syntax [78]. Therefore, AnaTempura is used to interpret and execute Tempura statements compliant with Tempura's syntax and formal semantics. Each Tempura operator has its semantically ITL-based equivalent and therefore inherits the semantics from it (see ITL semantics in [78]).

For referencing into the domain models using symbols s, the names of classes, roles attributes, and methods are used. These symbols are translated using the following rule:

$\mathcal{M}[\![s]\!] \rightarrow s$.

4.6.8 OCL and Tempura-based semantics

This section describes the essential operators for defining business rules as predicates in the context of invariants, pre- and post-conditions, and behavioural business rules. Moreover, an insight is given into how to symbolically address sets of objects which constitute the links between the APRIL statements and the domain model they are based on.

4.6.8.1 Basic operators

Table 4.2 depicts functions on basic types in APRIL and their translation into OCL and Tempura. Although APRIL strives to utilize natural language, these functions are kept in a simple mathematical style. It would be an inconvenient trade-off between understandability compared to clearness. As everyone should be aware of the meaning of the commonly accepted mathematical and logical symbols, there is no need for rewriting them in natural language.

4.6.8.2 Collections

Specifying business rules as logic statements requires reasoning about a universe of discourse which is formed by sets of objects. Like in OCL, APRIL's statements are based on type-based sets of objects that, on the one hand, hold certain predefined properties (also called attributes) and, on the other hand, can be related to each other using links. The formal rationale can be seen in the UML 2.4.1 specification [84]. Here is a brief overview:

A type to which an object is bound is described as a UML class (see [84]). A class can hold attributes and associations to other UML classes. Attributes can be used in subformulas (such as relational or mathematical operations) as constituents of predicate formulas or in other elementary set operations such as filters, union, or intersection

APRIL	OCL	TEMPURA	Type Signature
and	and	and	Boolean $\times$ Boolean $\rightarrow$ Boolean
or	or	or	Boolean $\times$ Boolean $\rightarrow$ Boolean
implies that	implies	not a or b	Boolean $\times$ Boolean $\rightarrow$ Boolean
not | it is not the case that	not	not	Boolean $\rightarrow$ Boolean
-(a)	-(a)	-(a)	Number $\rightarrow$ Number
+	+	+	Number $\times$ Number $\rightarrow$ Number
*	*	*	Number $\times$ Number $\rightarrow$ Number
/	/	/	Number $\times$ Number $\rightarrow$ Number
mod	mod	mod	Number $\times$ Number $\rightarrow$ Number
<	<	<	Number $\times$ Number $\rightarrow$ Number
<=	<=	<=	Number $\times$ Number $\rightarrow$ Number
>	>	>	Number $\times$ Number $\rightarrow$ Number
>=	>=	>=	Number $\times$ Number $\rightarrow$ Number
=	=	=	Number $\times$ Number $\rightarrow$ Number
<>	<>	~=	Number $\times$ Number $\rightarrow$ Number

Table 4.2: Basic and commonly used infix operators of the form a op b

operations (see also Table 4.2). Associations are used to indicate that objects of UML classes are related to each other, each playing a certain defined role in the scope of this individual relation. Links restrict the form of defining subsets of linked objects. The most general form of defining subsets on two or more source sets is the *Join* operation explained later, which is a special restricted form of a Cartesian product. In APRIL, a special form of *Join* operation is used for context-related set building. This is the "Collect" operator, indicated by the "." (see <NavigationPath> in Listing 4.6.4), which yields a set of objects that exclusively contains the objects of the type pointed to by the rightmost symbol of the collect chain.

The rationale of the collect operator is deduced in the following example using Figure 4.6. Here a UML class model with three different UML classes is defined, which are *Campus*, *Building*, and *Room*, which may serve as a simplified model of real world university campus. The concrete characteristic of the campus is reflected in the adjacent UML object model. A case in example is one top-level object named *University of*

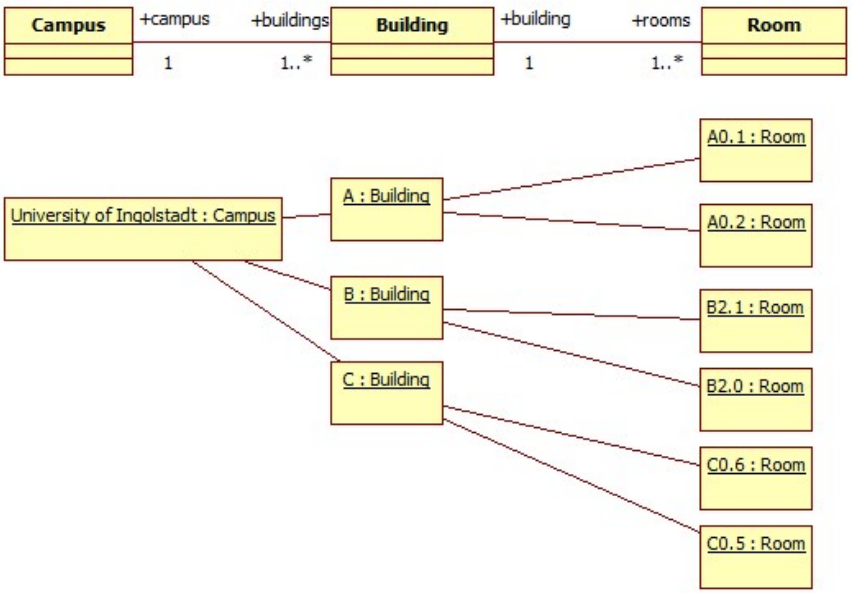


Fig. 4.6: Example of sets using UML classes and objects

Ingolstadt of type *Campus*, three objects of type *Building*, and six of type *Room*. The links in this example stand for a container-to-element relationship. A reasonable problem definition in this scope is to is to find out which objects are related to a superordinate object (e.g. which rooms a building consists of.) The meaning of the fundamental Cartesian product operator applied on the sets, e.g., like this

$$\text{Campus} \times \text{Building} \times \text{Room}$$

yields the set of 3 Tuples[2] presented in Table 4.3.

	A	B	C
A0.1	$\{U; A; A0.1\}$	$\{U; B; A0.1\}$	$\{U; C; A0.1\}$
A0.2	$\{U; A; A0.2\}$	$\{U; B; A0.2\}$	$\{U; C; A0.2\}$
B2.1	$\{U; A; B2.1\}$	$\{U; B; B2.1\}$	$\{U; C; B2.1\}$
B2.0	$\{U; A; B2.0\}$	$\{U; B; B2.0\}$	$\{U; C; B2.0\}$
C0.6	$\{U; A; C0.6\}$	$\{U; B; C0.6\}$	$\{U; C; C0.6\}$
C0.5	$\{U; A; C0.5\}$	$\{U; B; C0.5\}$	$\{U; C; C0.5\}$

Table 4.3: Cartesian product of the object population of Figure 4.6

In general, the Cartesian product is a very important operator in mathematic theories. However, the general operation applied to our problem neglects the link between the

[2]please note that the term "University of Applied Sciences Ingolstadt" is abbreviated as U

objects. The solution set that is of interest is a subset of the Cartesian product's result. Hence, more detailed consideration of the operation is needed that takes into account the individual links between the specified objects. Let a link λ between two objects (O_1 and O_2), with α_{0_1} as an attribute of object O_1 and α_{0_2} as an attribute of object O_2 be defined as a predicate that holds for:

$$(O_1.\alpha_1 == O_2) \wedge (O_2.\alpha_2 == O_1)$$

And let λ be the parameter operator of the Θ-Join operation (see Mishra et al. [73]), applied to the result of the Cartesian product above, the result for

$$\text{Campus} \bowtie \text{Building} \bowtie \text{Room}$$

yields the resulting tuples presented in Table 4.4.

	A	B	C
A0.1	$\{U; A; A0.1\}$		
A0.2	$\{U; A; A0.2\}$		
B2.1		$\{U; B; B2.1\}$	
B2.0		$\{U; B; B2.0\}$	
C0.6			$\{U; C; C0.6\}$
C0.5			$\{U; C; C0.5\}$

Table 4.4: Joined set version of Table 4.3

This comes very close to the answer desired at the beginning, but the result set contains a set of 3-tuples containing the objects of the types (classes) along the association path. The introductory problem statement required to yield a purely typed result set of the type pointed to by the rightmost symbol of the collect sequence. Hence, the collect operation

Collect(Campus , Building , Room)

casts any element of the tuple-based result set in Table 4.3 to a set of objects of type *Room*, which yields the set shown in Table 4.5.

	A	B	C
A0.1	$A0.1$		
A0.2	$A0.2$		
B2.1		$B2.1$	
B2.0		$B2.0$	
C0.6			$C0.6$
C0.5			$C0.5$

Table 4.5: Specially typed version of the set in Table 4.4

4.6.8.2.1 Associations The previous section used concrete sets of objects to explain the rationale of collection building using links. For the instance level of objects, it is reasonable to choose an operational description of the underlying mathematical principles such as the Cartesian product or the Θ-Join. However, the description of associated collections using these mathematical principles is done on a general level as the intention is to describe the rules for how to link the instantiated objects independently from the instances. The means to describe the afore presented collection construction is called association. The term and its semantics are introduced by the UML for the utilization in UML-class models. In the context of the association, each UML class at the respective association end reflects a certain partner with a defined role within the association. In UML, a role is denoted by a symbolic name, which is used as a parameter for the Collect operator. More details on referencing collections that are restricted by associations are presented in Section 4.6.8.2.2.

4.6.8.2.2 Symbolic referencing of collections For building sets, the symbols of role or class names occurring in a UML diagram are used, indicating the constituents of the addressed *source* set, which is used later in the semantic mapping tables.

$$[\![\, source \,]\!] \vDash [\![\, reference_1 \{.reference_N\} \,]\!] \vDash Collect(reference_1, \ldots, reference_N)$$

Here, *reference* stands for a role name of an associated class. The starting point is defined within the rule determining the reachable concepts (1 to N) for the first reference to the target set (N) to navigate to. A reference sequence along multiple associations and classes is often also referred to as navigation path.

4.6.8.3 Operations on collections

In Section 4.6.8.2.2 the basis of addressing sets are presented. In this section, the operators on sets can be used to apply additional functionality to the elements on the sets or operations on the set itself. Using these operators makes it necessary to take care of their type structure. This means that operators accept certain parameter types, which are abbreviated from the collection type and yield a result of a certain type. An indication on how the type characteristic of an operation is structured is given by the notation $TYPE_{param1}(\times TYPE_{paramN}) \rightarrow TYPE_{retVal}$. It says that the first parameter has type $TYPE_{param1}$, which is obligatory for almost any operation. Furthermore, optional additional parameters (paramN with N is a natural number) can be considered to be passed to the operation, whereas the corresponding type is $TYPE_{paramN}$. Eventually, any operation yields a return type indicated by $\rightarrow TYPE_{retVal}$.

The following tables, (4.6, 4.7, and 4.8), map APRIL operators to their OCL counterparts, whereas each pair follows an equal type structure.

APRIL operator or function	Corresponding OCL semantics
number of $iterator$ in $sourceSet$	⟦sourceSet⟧->count(⟦iterator⟧)
number of $sourceSet$	⟦sourceSet⟧->size()
sum of $sourceSet$	⟦sourceSet⟧->sum()
element at position pos in $sourceSet$	⟦sourceSet⟧->at(⟦pos⟧)
element at last position in $sourceSet$	⟦sourceSet⟧->at(⟦sourceSet⟧->size()-1)
aggregate *iterator* over *sourceSet* in *accumulator* as *Type2* using *body*	⟦sourceSet⟧->iterate(⟦iterator⟧ : ⟦Type1⟧; ⟦accumulator⟧ : ⟦Type2⟧ = ⟦initialValue⟧ | ⟦body⟧)
elementReference is type of *ClassName*	⟦elementReference⟧.isTypeOf(⟦ClassName⟧)
"if" $propExp$ "then" $anyExpression$ "otherwise" $anyExpression$	"if" ⟦propExp⟧ "then" ⟦anyExpression⟧ "else" ⟦anyExpression⟧
"this" | "this" $ClassName$	self

Table 4.6: Operations as partial functions with $ANY(\times ANY) \rightarrow [ANY|null]$

APRIL operator or function	Corresponding OCL semantics
every $iterator$ [, $iterator2$] in $sourceSet$ satisfies that $propExp$ every $iterator$ satisfies that $propExp$	⟦sourceSet->forAll(⟦iterator⟧ ⟦[, iterator2]⟧ | ⟦propExp⟧)
at least one $iterator$ in $sourceSet$ satisfies that $propExp$ at least one $iterator$ satisfies that $propExp$	⟦sourceSet->forAll(⟦iterator⟧ ⟦[, iterator2]⟧ | ⟦propExp⟧)
$object$ is in $sourceSet$	⟦sourceSet⟧->includes(⟦object⟧)
set is in $sourceSet$	⟦sourceSet⟧->includesAll(⟦set⟧)
$object$ is not in $sourceSet$	⟦sourceSet⟧->excludes(⟦object⟧)
set is not in $sourceSet$	⟦sourceSet⟧->excludesAll(⟦set⟧)
$sourceSet$ is empty	⟦sourceSet⟧->isEmpty()
$sourceSet$ is not empty	⟦sourceSet⟧->notEmpty()
$object$ is unique in $sourceSet$	⟦sourceSet⟧->isUnique(⟦object⟧)

Table 4.7: Operations on collections with $Collection \times Collection \rightarrow BOOLEAN$

APRIL operator or function	Corresponding OCL semantics
"each" iterator "in" source "where" propExp "each" iterator "where" propExp (short form)	⟦source⟧->select(⟦iterator⟧ | ⟦propExp⟧)
"each element in" source "without" propExp	⟦source⟧->reject(⟦propExp⟧)
"each element in" source "without" setReference	⟦source⟧->excluding(⟦setReference⟧)
"collection" setReference ["as set" | "as bag" | "as sequence" | "as ordered set"]	⟦setReference⟧->["asSet" | "asBag" | "asSequence" | "asOrderedSet"]()
"union of" sourceSet "with" setReference | sourceSet "including" setReference	⟦sourceSet⟧->union(⟦setReference⟧)
"union of" sourceSet "with" elementReference | sourceSet "including" elementReference	⟦sourceSet⟧- >including(⟦elementReference⟧)

"intersection of" sourceSet "with" setReference	⟦sourceSet⟧->intersection(⟦setReference⟧)
sourceSet "without" setReference	⟦sourceSet⟧ - ⟦setReference⟧
"reachable elements along" sourceSet | "deep collection of" sourceSet	context ⟦contextClass⟧ inv acyclic: self.successors()->excludes(self) context ⟦contextClass⟧ def : successors(): Set(⟦contextClass⟧) = self.⟦pathName⟧->union(self.⟦pathName⟧.successors())

for any natural numbers 1 <= n <= k and k > 2, where k is the number of sets to combine their elements:
"each" sourceSet {"," setReference}^{k} "combination"
	function ToOcl(n) {
	if n=k **then**
	[<$classReference_n$>.allInstances->forAll((i_n|<SyntheticAssociationClassName>->select(assoc| {assoc.@$Class_n$=i_n and assoc.@$Class_n$=i_n}^{n})]
	else
	[<$classReference_n$ >.allInstances->forAll(i_n | **ToOcl(n++)**)]
	}
	• **ToOcl(..)**: is a text replacement function that can be called and places text according to the control sequences in it. It takes a natural number for a parameter
	• **if .. then .. else**: is a typical control sequence for checking a condition and alternating between the then- and the else- path.
	• **num++** : is an operator to increment a natural number (num) by one.

sourceSet "of type" *className*	⟦sourceSet⟧->select(element | element.isTypeOf(⟦className⟧))

Table 4.8: Operations on Collections with $Collection \times Collection \rightarrow Collection$

4.6.9 Semantics for behaviour rules

In APRIL, atomic formulas are the basic building blocks for business rule statements. Each atomic formula can be directly mapped to a semantic pendant in the respective target language and does not have to be decomposed any further by the parser, which is why the term *atomic* was chosen. Atomic formulas can be customized or added, which makes it possible to extend the expressive power in the context of a certain domain. This section gives some example operators to enhance APRIL's power towards a group of domains which deal with behaviour. Therefore, Table 4.10 states some ITL operators, with an appropriate verbalized version in APRIL along with the Tempura operator, reflecting the counterpart within the executable subset. Please note that in the context of atomic formulas, the presented APRIL versions are a straightforward verbalization and may not consider a general linguistic correctness of their valid combinations. Again, a decent natural language reading can be supported by means of decomposition. The type structure for all operators is $(ANY(\times ANY)) \rightarrow BOOLEAN$. There are also operators that do not require a parameter, e.g., $skip$ or $empty$ since they test certain properties of the interval.

In Table 4.10, there are some examples of the basic set of operators describing behaviour which a user might want to define. Some more operators along with their semantics are presented by Moszkowski in [77].

APRIL atomic formula	Tempura statement	ITL
true	true	$true$
false	false	$false$
it is not the case that p	not $[\![p]\!]$	$\neg[\![p]\!]$
A is followed by B	$[\![A]\!]$; $[\![B]\!]$	$[\![A]\!]$; $[\![B]\!]$
A holds repeatedly	chopstar $[\![A]\!]$	$[\![A]\!]^*$
interval of length 0	empty	$empty$
interval of length x	len x	len x
it is always the case that p	always $[\![p]\!]$	$\Box[\![p]\!]$
it is sometimes the case that p	sometimes $[\![p]\!]$	$\Diamond[\![p]\!]$
next time p	next p	$\bigcirc[\![p]\!]$
p or q	$[\![p]\!]$ or $[\![q]\!]$	$[\![p]\!] \vee [\![q]\!]$
A and B (parallel composition)	$[\![A]\!]$ and $[\![B]\!]$	$[\![A]\!] \wedge [\![B]\!]$
q and p (Boolean)	$[\![q]\!]$ and $[\![p]\!]$	$[\![q]\!] \wedge [\![p]\!]$
p implies that q	$[\![p]\!]$ implies $[\![q]\!]$	$\neg[\![p]\!] \vee [\![q]\!]$
it is not the case that A	~A	$\neg[\![A]\!]$
interval of lenght 1	skip	$skip$
var is constant	stable A	stable A
more than one state	more	more
until A	halt A	$true$;A
eventually A	fin A	fin A
e_1 gets e_2	e_1 gets e_2	$\Box(more \rightarrow [(\bigcirc e_1) = e_2])$
at least one *def* in *defs* satisfies that A	exists *defs* : A	existential quantification, see [78]
every *def* in *defs* satisfies that A	forall *defs* : A	universal quantification, see [78]
if p then A otherwise B	if p then A else B	if p then A else B
$=, >, <, >=, <=, <>$	$=, >, <, >=, <=$,~=	conventional relational operators, see [78]

Table 4.9: A version of atomic formulas covering behaviour

In Table 4.10, there are some examples of some useful verbalizations of combined operators that can extend the basic set of operators of Table 4.9 due to making APRIL statements terser.

APRIL atomic formula	Tempura statement	ITL formula
it must never occur that p	always { not(p) }	$\Box\neg p$
p is true at the end	sometimes { empty } and fin p	$finite \wedge fin\, p$
q is sometimes stable	sometimes { always { q }}	$\Diamond\Box q$
p inverts q permanently	always { p implies (sometimes { always { not(q) }}) }	$\Box(p \rightarrow \Diamond\Box\neg q)$

Table 4.10: Some user-defined atomic formulas for behaviour

4.6.10 APRIL's target languages

APRIL makes use of the logical frameworks OCL and Tempura to underpin its language constituents with a well-defined semantics. Both languages are briefly introduced in the subsequent sections.

4.6.11 OCL

As part of the UML, OCL 2.3.1 is the target language for APRIL invariants and pre- and post- conditions. For the sake of brevity, a rudimentary introduction to OCL is presented because it is a specified standard. The interested reader should consult the literature on OCL. The specification of OCL 2.3.1 can be found in [85].

OCL restricts UML-class models using predicate logic and operations on sets. Arithmetic, Boolean, and relational operators are used in a conventional way. Existential and universal quantifiers allow quantifying on propositions holding on an object population derived from a class model. In order to give an idea of the OCL syntax, Listing 4.6.7 presents a translation into OCL of the example in Listing 4.6.5 and the definitions (D.1)-(D.3) of Listing 4.6.6, both adopted from Section 4.2. The OCL also supports mechanisms for decomposing statements that can be used as translation pendants of the respective APRIL definitions or local variables, which makes the definition of the translation templates easier on the one hand and, on the other hand, supports the readability of the OCL statements, e.g., for debugging purposes.

```
1 Invariant rule1 concerns Customer is defined as
2 All premium customers who order a special offer must pay 0 EURO
    for the shipment of that order.
```

Listing 4.6.5: Top-level rule from Section 4.2.2

```
Definition All (customers as Collection of Customer) who order (
   products as Collection of Product) must pay (price as Number)
    EURO for the shipment of that order yielding Boolean is
   defined as
every customer satisfies that every "ordered product" satisfies
   that shipment.fee = price
with
"ordered products" (orderer as Customer) is defined as each
   product where product.order.customer = orderer.

Definition special offer yielding Collection of Product is
   defined as
each customer in all instances of Customer where customer.
   AverageAnnualTurnover  $>$ 20,000.

Definition special offer yielding Collection of Product is
   defined as
each product in all instances of Product where product.
   IsSpecialOffer.
```

Listing 4.6.6: Definitions D.1, D.2 and D.3 from Section 4.2.2

```
context Customer inv:
All_customers_who_buy_products_must_pay_price_for_shipment(
Customer::premium_customers(),
Product::special_offers(),
0)

context Customer def:
```

```
All_customers_who_buy_products_must_pay_price_for_shipment(
customers    : Collection(Customer),
products     : Collection(Product),
price        : Real)                    : Boolean =
customers->forAll(customer | products->select(product | product.
   order.customer = product)->forAll(orderedProduct |
   orderedProduct.shipment.price = price))

context Customer def:
premium_customers() : Collection(Customer) = self.
   AverageAnnualTurnover > 20,000 EURO

context Product def:
special_offer():  Collection(Product) = self.IsSpecialOffer =
   true
```

Listing 4.6.7: Translation of rule in Listing 4.6.5

4.6.12 Tempura

Tempura is an executable subset of Interval Temporal Logic (ITL) [74, 78]. ITL enhances predicate calculus with a notation of discrete time, expressed by separated states and associated operators. A key feature of ITL and Tempura is that the states of a predicate are grouped together as nonempty sequences of states called intervals σ_{plus}. For example, the shortest interval of states σ on a predicate can be represented by $< s >$, where s is a state. Please note that here the length $\sigma := |\sigma| = 0$, which is generally the number of states in σ minus 1. The semantics of ITL keeps the interpretations of function and predicate symbols independent of intervals. Thus, well-known operators such as {+, -, *, and, or, not,...} are interpreted in the usual way. The characteristic operator for ITL

is the operator *chop (;)*, which says that a prefix subinterval is *followed by* a suffix subinterval. Both subintervals share one state "between" them. Conventional temporal logic operators such as *next* ($\bigcirc$) and *always* ($\Box$) examine an interval's suffix subintervals, whereas chop splits the interval into two parts and tests both. Furthermore, Moszkowski [74] shows how to derive operators such as always and sometimes from chop. In ITL, the formula $w := w_1; w_2$ is true if $\mathfrak{I}_{\langle \sigma_0 .. \sigma_i \rangle} [\![w_1]\!]$ and $\mathfrak{I}_{\langle \sigma_i .. \sigma_{|\sigma|} \rangle} [\![w_2]\!]$ are true in the respective sub-formulas. See further details in [74]. Note that w_1 and w_2 share the same subinterval σ_i.

4.6.12.1 Introductory examples

Some examples from [74], can be found in the following:

σ	P	R
s	1	2
t	2	1
u	3	1

The length of interval σ is expressed by $|\sigma|$ and is defined as the number of the states in σ minus one. Thus, in our example, $|\sigma| = 2$.

The following formulas on the predicates P and R are true on the interval $< stu >$:

- $P = 1$. The initial value of P is 1.
- $\bigcirc(P) = 2$ and $\bigcirc(\bigcirc(P)) = 3$. The next value of P is 2 and the next to next value of P is 3.
- $P = 1$ and P gets $P + 1$. The initial value of P is 1 and P gets increased by 1 in each subsequent state.
- $R = 2$ and $\bigcirc(\Box(R)) = 1$ The initial value of R is 2 and R is always 1 beginning from the next state.

- $P \leftarrow 1 \; ; \; P \leftarrow P + 1 \; ; \; P \leftarrow P + 1$. The formula $e_2 \leftarrow e_1$ is true on an interval if $\sigma_0(e_1)$ equals $\sigma_{|\sigma|}(e_2)$. Thus, $\leftarrow$ is called a temporal assignment.

Tempura is chosen for the semantic underpinning behaviour because it is able to model operations lasting over multiple state transitions, which would not be possible with a single pair of OCL pre- and post-conditions.

4.6.12.2 Translation example

The following example shows a translation of a behavioural APRIL rule (see Listing 7.14) into Tempura (see Listing 4.6.8) based on an example in Section 7.6.2.

```
1
2  /* example historical data recorded during the test scenario execution */
3  define DataRecord = [ [00,0,0,1,1,12],
4                        [00,0,0,1,1,12],
5                        [10,0,0,1,1,12]].
6
7  /* auxiliary function for accessing the data record */
8  define get(i, idx) = {
9    DataRecord[i][idx].
10 }.
11
12 /* array index list for accessing the history data array */
13 define CAN_ZV_SCHL_A = 0.
14 define VEHICLE_LOCKED = 1.
15 define KL_START = 2.
16 define S2_FT_RIEGEL = 3.
17 define S2_T_RIEGEL = 4.
18 define BatteryVoltage = 5.
19
20 /* framework function defined with a specific semantics. */
21 define aHasNoEffectOnB (A, B) = {
22  always{A implies {sometimes{always{not(B)}}}}
23  }.
24
25 /* translation */
26 /* run */
27 define tryStartEngineAtLowBattery () = {
28 exists I : {
```

```
29 define lowBattery() = {always{ get(I,BatteryVoltage) < 9}} and
30 define detectionOfTheOpeningSignal() = {get(I, CAN_ZV_SCHL_A) = 10} and
31 define doorLockState() = {get(I,S2_FT_RIEGEL) = 1 and get(I,S2_T_RIEGEL) = 1} and
32 define engineStarts() = {get(I,KL_START) = 1} and
33 {
34  {
35   lowBattery() and
36   aHasNoEffectOnB(detectionOfTheOpeningSignal(),doorLockState())
37  }
38  implies
39   engineStarts()
40 } and
41 I gets I+1 and
42 len |DataRecord|-1 }}.
```

Listing 4.6.8: Tempura pendant of rule 7.14

The Tempura program code analyzes a data record of an actual test scenario gathered during the execution of the actual software system. Being able to process the relevant data record, the format of the data has to be compliant with the Tempura syntax, too. In line 3 of Listing 4.6.8 a two dimensional array has been chosen to achieve this. Therefore, this array contains a sequence of tuple elements. Each tuple holds the value of a variable representing a model concept of a certain state in which they were recorded. In order to add more readability to the generated Tempura code the lines 13-18 introduce a mapping of concept names to indexes for the tuple items in the data records. The auxiliary function *get* is used to access a certain state variable from the records. Additional auxiliary code in the lines 41 and 42 ensures that the Tempura interpreter uses each tuple from the record array a set of states. Here, the *len* operator determines the size of the interval, which shall correspond to the size of the recorded data, whereas the *gets* operator increments an iterator variable *I* for each state, allowing to access a tuple in the array corresponding to the active state. This auxiliary code is necessary to get the actual business rule running, which is transformed as follows.

Within the main function *tryStartEngineAtLowBattery* (indicated by the *run* comment, lines 26 et seq.) the actual business rule is placed. Local functions and variables in APRIL are translated as Tempura functions in the scope of the main function (lines 29-32) introduced by the *define* key word. Note that any expression within a Tempura formula must be concatenated by propositional operators such as *and*, *or* etc., which is why each of the example local functions is concluded by an *and* operator. Atomic formulas and basic operators (e.g. *exists*, *and*, *implies*, etc.; see lines 28, 35 and 38) are translated one by one according to the rules presented in Table 4.9.

4.7 Summary

This chapter introduced the new language APRIL and gives an impression of the language features of the overall APRIL framework. The language features are

- Presentation of the syntax using examples.
- Decomposition by APRIL definition and local variables
- Mixfix notation as a means to describe sentence patterns to cluster logical expressions to appear in a syntactical form close to natural language.
- Formulating basic logical operators as atomic formulas
- Usage of common constraints as default atomic formulas for expressing frequently used logical formulas concisely.
- Some useful rules for syntactic sugaring.
- Semantics of the Invariant, pre-/post-condition, and behavioural language constructs.

The framework aspects presented in this chapter are:

- Description of a mechanism for adding new custom atomic formulas to enhance the expressive power of the APRIL language.

To the best of our knowledge, the introduction of mixfix notation for APRIL definition headers is a novel approach for decomposing large business rules and making them more understandable to human readers. A second novel aspect concerns the framework, which allows extending the language itself at the root level using compiler building mechanisms. Both aspects contribute to the tendency that in practice, frequently used logic is first cast into reusable artefacts and if approved and generic enough, even find their way into the language itself, e.g., as operators.

Chapter 5

Processing, implementation, and integration of APRIL

5.1 Introduction

In this Chapter, the technical aspects of the transformation of APRIL statements to OCL and Tempura are explained. As mentioned in the previous sections, the core aspect of APRIL is the ability to specify sentence patterns in mixfix notation, which is particularly suitable for specifying user-defined functionality with a close natural language representation. Other important aspects presented here are type checking and scoping, which allow providing a clear structure to the semantics of the specified APRIL statements and helps to keep the compiler logic as simple as possible. Additionally, the specification of target language templates for the APRIL code generator is explained. This can also be used to exemplify the extension of APRIL by user-defined atomic formulas. The chapter concludes with a presentation of the APRIL compiler architecture and its input and output artefacts, which is tied up in Figure 5.10.

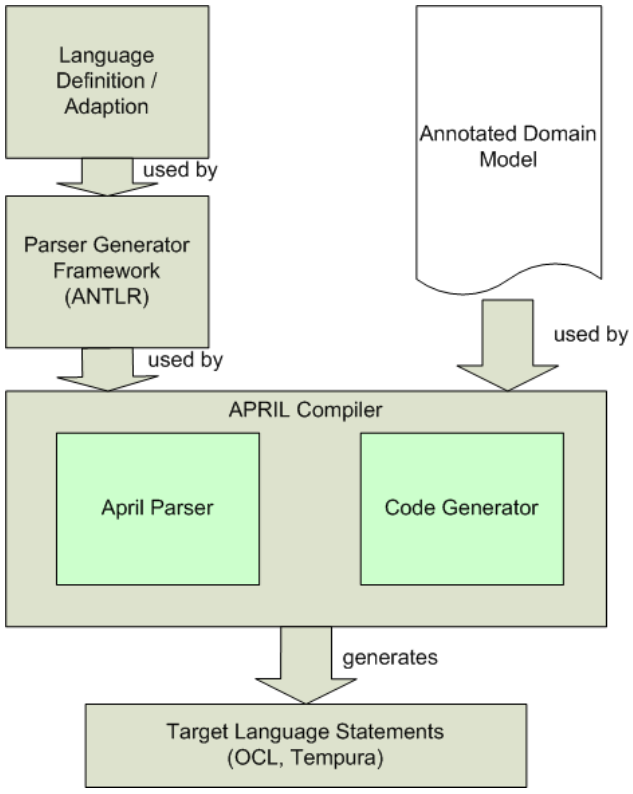


Fig. 5.1: Sketch of APRIL compiler components and input artefacts

For the better understanding of the different steps, concepts, and their interrelations,

Figure 5.1 gives initially a brief overview of the concepts presented in Figure 5.10. The APRIL compiler has some generated parts that are specified in the formal APRIL grammar, which is used by the parser generator ANTLR (Version 3.0) to generate parts of the APRIL parser. The parsing logic for mixfix patterns is handcrafted and will be explained later in this chapter. For the translation of the annotated domain model, the code generator traverses the output of the APRIL parser and replaces the nodes of the AST by the target language templates defined for each APRIL statement. Generating the code for the respective target language follows an inductive approach in which each parameter of the AST node is placed into the dedicated placeholder positions of the target language template. The results of the code generator are OCL or Tempura statements.

5.2 Parsing mixfix signatures

One of the core features that contributes significantly to the linguistic well-formedness of an APRIL expression is the ability to define operators with a mixfix syntax. Mixfix operators look intriguingly natural to human readers, which is validated in the case study in Section 8. The reader also can get a feeling in some of the example APRIL definitions of the previous sections. Supporting mixfix definition signatures in the syntax makes the implementation of the parser logic more complex. In the following sections, it is shown that sophisticated features have to be utilized. For a better understanding, please consider the running example, starting from an APRIL definition in Listing 5.2.2, known from Listing 4.2 in Section 4.2.3. The rule in Listing 5.2.1 consists of one top-level definition (D1) (see Listing 5.2.2) and two embedded definitions (D2 and D3) (see Listing 5.2.3 and 5.2.4). In this example, nesting is uncritical, hence the definition signatures of D2 and D3 are the constants *premium customers* and *special offer*, which is non-ambiguously resolvable to the parser.

However, APRIL's context-free grammar definition for mixfix definition signatures (Non-terminal: *MNParamDefinition* in Section 4.6.6) allows constructing valid syntax,

imposing a potentially unresolvable ambiguity. Consider the following simple abstract example of possible definition signatures. (Without strong typing, T1 and T2 are types determined at runtime, C stands for constant, and V_x stands for a placeholder of a variable, whereas V is a reference that "fills" the placeholder.) Given the two definitions in Listing 5.1, a valid statement as in Listing 5.2 is possible.

Def1: V_{11} C_{11}, with Def1 of type T1 and V_{11} of type T2
Def2: C_{12} V_{12}, with Def2 of type T2 and V_{12} of type T1

Listing 5.1: Abstract definition description of forming ambiguous statements

C_{12} V C_{11}

Listing 5.2: Ambiguous statement based on improper definitions

Without additional information, it is not clear if V is linked with V_{12} or V_{11} and therefore if Def1 is nested in Def2 or vice versa, which can lead to a different semantics. To overcome this problem, APRIL uses static strong typing that requires specifying a type to variables and definitions at design time. This means that it is obligatory to define specifications for variables, definitions, and operations with a predefined type. In general, strong typing also makes it possible to detect design errors early in the design phase. A consequence is that a statement has to comply with the type definition of the hosting operator or expression body. In the case of the example, C_{12} V C_{11} is resolvable depending on the return type expected by the parent operation, which is different in contexts where either T1 (Def1 nested in Def2, short Def1(Def2)) or T2 (Def2(Def1) is expected; given that T1 $\neq$ T2). Although typing allows non-ambiguous resolution, depending on the context, it may be misunderstood by a human reader. Therefore, APRIL offers two further syntax-based mechanisms to make the nesting of operators explicit. The recommended mechanisms are to use decomposition (e.g., using local variables or transfer logic to other definitions) for a better readability of the overall statement. The fallback mechanism is to use round brackets as defined in APRIL's Grammar (see Section 4.6.4).

Please note that as mentioned in the earlier section, each definition has a type structure (see also Figure 5.2) that always defines a return value and requires input type definitions

for parameters. A rule (at the top level) is always defined as a predicate, which means that is must yield a return value of type Boolean, which restricts the result elements to be an element of the set { $true$, $false$ }. Any subordinated definitions or formulas can be typed arbitrarily. As mentioned, APRIL uses static strong typing. In the context of typing, strong means that the type of a concept is fixed during compile time and cannot be changed during run time. Strong typing allows detecting design errors by the compiler, which is early enough to avoid time-consuming debugging at runtime, thus constituting a gain in efficiency. Static typing, on the other hand, makes it mandatory to define a type to a concept at design time, which cannot be changed at runtime.

The principle of static strong typing for supporting the parser logic to correctly understand mixfix definitions, along with the mechanisms resolving mixfix signatures, will be explained in the following. The interested reader can find some further information on static strong typing and its importance to programming in the relevant literature such as Liskov's paper [62].

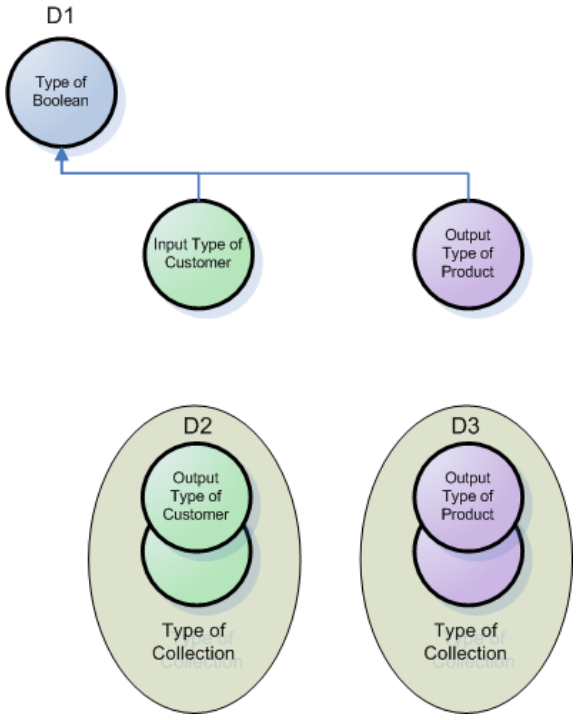


Fig. 5.2: Type structure of the definitions D1-D3

```
1 Invariant rule1 concerns Customer:
2 All premium customers who order a special offer must pay 0 EURO
    for the shipment of that order.
```

Listing 5.2.1: Example taken from Listing 4.1 in Section 4.2

According to the definitions in section 4.2, the APRIL rule in 5.2.1 is structured by three definitions (see listings 5.2.2, 5.2.3, and 5.2.4).

```
1 Definition All (customers as Collection(Customer)) who order (
    products as Collection(Product))
2   must pay (price as Number) EURO for the shipment of that order
3    yielding Boolean
```

Listing 5.2.2: Definition signature of (D1) in Section 4.2

```
1 Definition premium customers yielding Collection(Customer)
```

Listing 5.2.3: Definition signature of (D2) in Section 4.2

```
1 Definition special offer yielding Collection(Product)
```

Listing 5.2.4: Definition signature of (D3) in Section 4.2

pathNameOrDefinitionCall::= {α_0}modelReference
|{α_1} definitionCall | ...;
definitionCall::= ID (ID | pathNameOrDefinitionCall)* ;

Listing 5.3: Grammar snippet for APRIL mixfix definitions (see also 4.3)

A definition signature consists of an arbitrary sequence of constants and placeholders for parameters, reflected by the formal grammar snippet in Listing 5.3. The example

definition D1 in Listing 5.2.2 is introduced by the constant *All* represented by the *ID* definition of the grammar, allowing a character sequence defined by the regular expression *ID ::= ([a-z][A-Z])([a-z][A-Z][0-9][_])**.

The regular expression is a rule that prescribes the form of a character sequence. APRIL's ID non-Terminal must start with at least one lower or uppercase letter (indicated by *([a-z][A-Z])*), followed by an arbitrary sequence of either lower or uppercase letters, numbers, or the special character "_". An empty sequence is also allowed, indicated by the Kleene star "*" at the end of the second part of the regular expression. The second constituent of the signature is a placeholder for a parameter that accepts the symbols representing sets of, e.g., *Customer* (in the context of the example: "Collection(Customer)"). If the definition is used in a business rule, the placeholders are supposed to be filled with a symbol that represents a concept of that type required by the parameter definition, which is similar to programming languages (e.g., Java). Symbols can be defined within the rule itself as local variables, as APRIL definitions, or as parts of the basic UML-class model (navigation paths) that the respective business rule is based on. The typing for local variables does not have to be made explicit as is deduced by the compiler, which uses the type of the operator used in the rule body in order to assign a type to the variable. Sets representing navigation paths consist of concatenated names of rule-ends of associations between classes (see Section 4.6.8.2.2). Symbols are supposed to be unique identifiers within the scope of an APRIL statement, that allows addressing a certain semantic concept (e.g. a set of a certain type). As it is not always possible to keep a symbol unique throughout the entire specification, APRIL, like other languages, utilizes scoping. Scoping helps to restrict the set of (equal) symbols visible to an APRIL statement in order to yield exactly one semantic concept for each visible symbol. Moreover, scoping guides the resolution mechanism to find the intended symbol by determining a starting point. If equal symbols with equal return types occur more than once and are visible by a statement, scoping provides rules to the parser logic for prioritisation of the visible symbols. Therefore, APRIL distinguishes five precedence levels (see Figure 5.3).

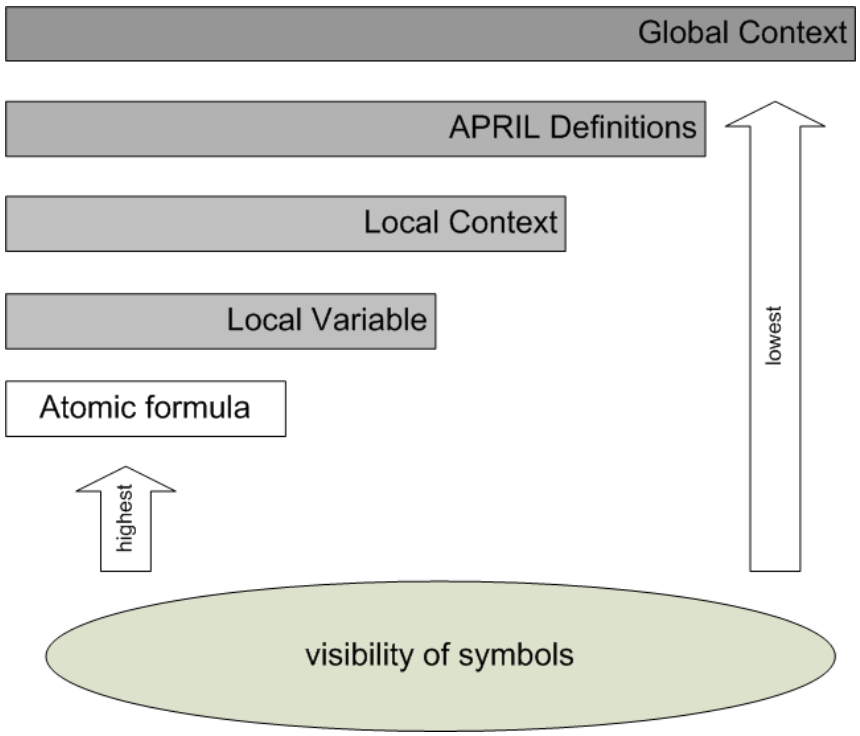


Fig. 5.3: Visibility of symbol kinds

In the following, some valid modifications to the UML-Model example rules are made.

- Property *IsSpecialOffer* of class *Product* is renamed to *specialOffer*
- Add property *Price* of type Number to the class *Product*.

Considering the renamed and newly introduced property, it is obvious that a mechanism to distinguish between unique symbols is needed. Consider the definition in Listing 5.2.5 combined with the following modifications to the underlying UML-class model:

```
1 Definition specialOffer yielding Collection(Product)
2 is defined as
3 each product in all instances of Product where
4   product.specialOffer = true and CustomerIsAustrianAdult
5 with
6 CustomerIsAustrianAdult is defined as shipment.Destination = "
    Austria" and order.customer.DateOfBirth > "1998.01.01" .
```

```
7
8
9  Invariant ScopingExample concerning Product
10 it is not the case that specialOffer and Price > 1000 .
```

Listing 5.2.5: Simple scoping example using local context and APRIL definition

In the invariant *ScopingExample*, the symbol *specialOffer* matches the property of the context class and the signature of the definition above. The precedence for selecting the concept prescribes that that concept is relevant with the higher visibility (see Figure 5.3). In the case of the example, there are two visibility levels, *local context*, and *APRIL definitions*, competing. According to APRIL's precedence rules, the parser logic selects the concept of the *local context* to be relevant for the rules semantics. It is obvious that such resolution mechanisms have to be properly defined, which is done by a suitable grammar definition and additional logic incorporated in the parser as semantic annotations. The following gives an explanation of the relevant APRIL grammar definition.

According to the Grammar definition of Section 4.6.1, the following production rule *pathNameOrDefinitionCall* is applied when symbols pointing to a typed concept (e.g., the definition of Listing 5.2.5) have to be resolved.

pathNameOrDefinitionCall::= $\{\alpha_0\}$ *pathname* | $|\{\alpha_1\}$ *definitionCall* ;

Listing 5.4: Grammar snippet for APRIL mixfix - Definitions (see also 4.3)

Here a distinction is made between a reference to a concept of the UML Model (in the scope of the local or global context), or a definition or local variable has to be recognized. Please note that atomic formulas are resolved differently. They are part of the grammar and get represented by individual production rules. The usage of the definition or a local variable implies that the constants and the variable placeholder of their signatures are stated in the correct, predefined order. This order is reflected by the *definitionCall*

production rule, allowing to put constants (indicated by the terminal ID) around symbols of typed concepts acting as parameters. For reducing the technical complexity of the implementation, a *definitionCall* has to start with a constant (ID).

definitionCall::= ID (ID | pathNameOrDefinitionCall) ;*

Listing 5.5: Grammar snippet for APRIL definition calls (see also 4.3)

The mechanism for how the mixfix parser works is explained later. Like other parser generator frameworks, the utilized ANTLR V3.0 starts the interpretation of a list of APRIL statements with a lexical analysis [2]. This first step applies regular checks to the statements and transforms them into a TokenStream, interpretable by the generated LL-parser [2]. The parser is an automaton that transforms a TokenStream into an abstract syntax tree. This works for the expression body of the APRIL rule (Section 5.2.1) and also any expression body as follows.

The first step is to find out which logical concept to apply. Therefore, the parser uses scope-related information consisting of lists of regular expressions generated from the definitions, local members (either variable symbols or function signatures), local and global contexts, and the atomic formulas against which to match the expression.

The TokenStream in Listing 5.6 is matched against the generated regular expression patterns visible within the context of the rule. The first match of a concept in the scope level with the highest visibility determines the type of node – representing the found concept – to append to the abstract syntax tree. In the case of the example TokenStream, the first match occurs with a pattern extracted from the APRIL definition (see Listing 5.2.1).

Once a pattern is matched and a concept found, the appropriate node for the abstract syntax tree is generated according to the grammar. In this example, the node for the

All | premium customers | who | order | a | special offer | must | pay | 0 | EURO
for | the | shipment | of | that | order | .

Listing 5.6: Token representation of the APRIL statement in Listing 5.2.1

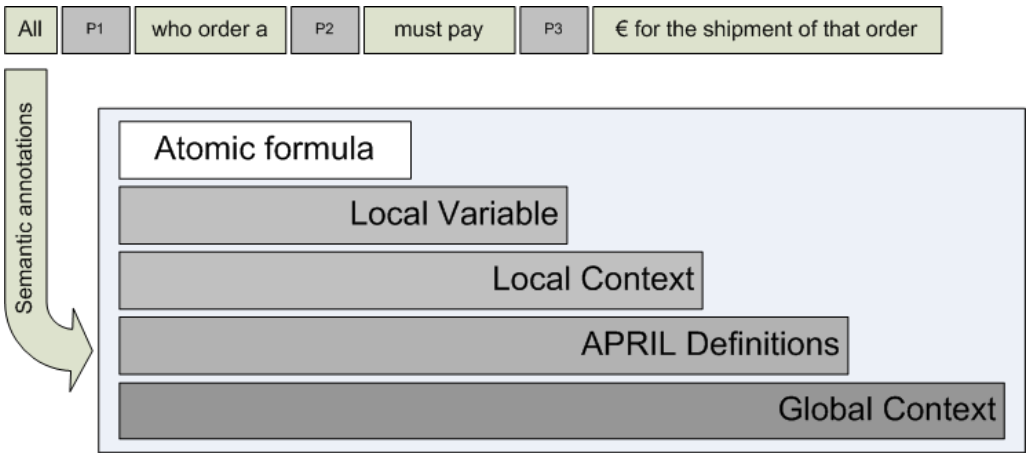


Fig. 5.4: Pattern matching of the "All premium customers..."-example rule against the scope stack

definition call of D1 is generated. At that time, the TokenStreams representing the symbols for the parameters of D1 are unresolved, and are hence subject to the subsequent resolution step. Figure 5.4 shows that the parameter P1 that is typed as *Collection of Customer* is in this instance of D1 configured with the symbol *premium customer*, which is the symbol for another definition, namely D2. P2 typed as *Collection of Product* is configured by *special offer* being a symbol for D3. P3 is a constant of type *Number*. According to that resolution, the node of the abstract syntax tree representing this rule looks like in Figure 5.5.

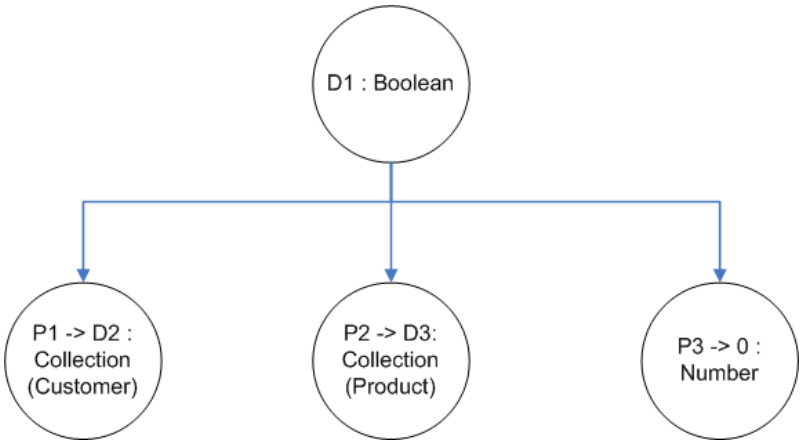


Fig. 5.5: Abstract syntax tree representation of D1

Semantic annotations need to be utilized for correctly detecting mixfix signatures in the TokenStream and producing the intended abstract syntax tree image (as in Figure 5.5).

As defined in the grammar in Listing 5.3, a semantic annotation is used as a predicate that guides the resolution mechanism of the automaton. In the case of α_0 (see also algorithm of Listing 5.2.6), the underlying logic tries to match the subset of the TokenStream against the visible entities contained in the scope stack (see Figure 5.4). Once the current token to interpret matches one of the visible symbols in the list of domain model types, represented by a role or property name, a pathname concept is detected. This means that $alpha_0$ yields *true* and a pathname node is added to the abstract syntax tree. This is in analogy with the different kinds of concepts that are held by the scope stack. The levels *local variable* and *local context* of the stack are individually adapted to the current state of the automaton. The elements in the set of the local variables level are individual to each rule or definition. E.g., the definition named *specialOffer* in Listing 5.2.5 contains the local variable *CustomerIsAustrianAdult* in its local variable definition context. Whereas, the local context of the invariant rule (also in Listing 5.2.5) reflects the entry point into the domain model at the type with the symbolic representation *Product*. Finally, any APRIL definition is visible in the contexts of APRIL rules and other definitions. As mentioned before, each list in the scope stack has a different visibility to a statement, imposing a precedence for potentially matching symbols in different scope levels. For example, if there would be an APRIL definition competing with a local definition, the parser would select the local definition as it has the higher precedence (visibility) compared to the definition (algorithm of Listing 5.2.6).

```
input string[] tokenStream, ScopeStack visibleSymbols,
output bool isPathName
begin
  if tokenstream matches any domainModelSymbol in visibleSymbols then
    isPathName = true
  end if
  isPathName = false
end
```

Listing 5.2.6: The annotation α_0

Determining that the annotation α_1 holds is more delicate. The reason is that a subset of a potentially infinite TokenStream has to be extracted against which to match the APRIL definitions, which is especially difficult with definitions ending with a variable. In this case, the parser cannot decide if the definition ends at a certain point after the last constant, as any subsequent token is only an ID terminal amongst ID terminals. Solving this problem imposes restrictions on how to define statements that may lead to unnatural but valid syntax. However, this is considered to be a minor limitation for defining APRIL definitions with mixfix syntax because it can be bypassed elegantly by factoring out a complex statement into a new APRIL definition or a local variable. Nevertheless, finding the correct ID token that terminates a definition call has to be solved anyway, which is outlined as follows.

```
input string[] tokenStream, ScopeStack visibleSymbols,
output bool isDefinitionCall
begin
  if tokenstream matches any definition_regex in visibleSymbols then
    isDefinitionCall = true
  end if
  isDefinitionCall = false
end
```

Listing 5.2.7: The annotation α_1

According to the grammar, definition calls are used in expression bodies of an APRIL definition or rules. Hence, the main use case for the application of definition calls is that of a top-level definition that embeds several other definitions as parameters (e.g., see the running example in Listing 5.2.1). Therefore, it is only necessary to determine the statements' bodies' ends, terminating at the non-terminals either "with" or "." followed by a newline character. Remember, the *with* nonterminal indicates that the definition section for local variables starts, whereas the "." (dot) terminates a rule or definition, since both are followed by a newline character. A not so trivial use case is when definition calls are used as parameters of other operators. In this case, the aforementioned strategy is not fully applicable. Here, the fallback strategy of utilizing brackets (see grammar in Section 4.6.4) has to be applied for boxing the definition call and thus indicating the beginning and end of a definition call. Resolving the relevant closing bracket, the APRIL parser uses a simple linear automaton that recognizes opening brackets of other embedded expressions and matches them against their corresponding closing pendants until the introductory closing bracket of the actual definition call is found. As mentioned before, using brackets is not the preferred way to obtain precedence for embedded operations in APRIL but they are nevertheless necessary for the user-defined structuring of grammatically valid statements. In certain cases, there may be syntactic constructs that cannot be expressed unambiguously without obtaining explicit precedence.

Once a TokenStream that is potentially a definition call is isolated, it is possible to further traverse through the production rule (*pathNameOrDefintionCall* in Listing 5.5) and apply the next semantic annotation expressed by α_1 in the relevant grammar snippet in Listing 5.3.

Again, the semantic annotation α_1 can be seen as a predicate, which is *true* if the following holds for a TokenStream (see also algorithm of Listing 5.2.7). Let T_s^* be the unresolved but isolated (see the previous section) TokenStream of an APRIL statement (such as Listing 5.6), which the parser has to interpret for the production rule *pathNameOrDefinitionCall* (from Listing 5.3). The star "*" in symbols like T^* shall

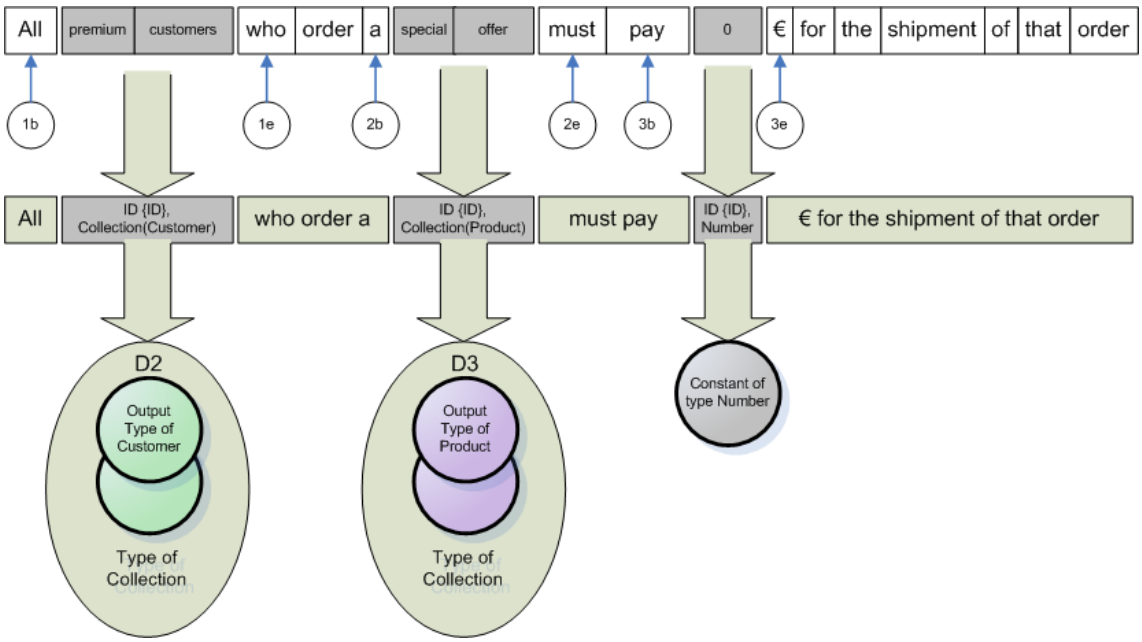


Fig. 5.6: Extracting embedded child definition calls by constant elimination based on regular expression of parent definition

indicate that a non-empty token set with a natural number of tokens is represented.

Then, at this stage of the automaton, the current token to interpret shall be T_0. The end token T_e of a (potential) definition call at the uppermost call level, which points to an APRIL definition with mixfix syntax and nested definition calls (such as in our example from Listing 5.2.1 and the tokenized version of Listing 5.6) is indicated by a terminal symbol.

Then the relevant TokenStream T^*_{call} to match against the previously defined APRIL definitions D^*_{RegEx} is the subset $T^*_{call} := \{T_0, \ldots, T_e\} \in T^*_s$. Each signature of an APRIL definition D_{Def}, whether in mixfix form or not, is transformed into a regular expression D_{RegEx}, where a constant in the definition $Const_{D_{Def}}$ is transformed into a terminal in the regular expression $Const_{D_{RegEx}}$, and a variable placeholder of the definition $Var_{D_{Def}}$ is transformed into a 2-tuple of D_{RegEx}, namely $(ID^{*>0}; Type)$, which is a non-empty set of ID tokens and type information ($Type$) that is resolved later on. However, the type information is only relevant for overloading, which is not covered here. The symbol $ID^{*>0}$ is the short form of the EBNF form of $ID\{ID\}$. Resolving the correct APRIL definition based on T^*_{call} requires that $\exists d^{RegEx}_n \in D^*_{RegEx} : match(d^{RegEx}_n, T^*_{call})$. This

means that a certain single regular expression $D_{RegEx} \in D^*_{RegEx}$ matches T^*_{call} in order to allow the production of the rule *pathNameOrDefinitionCall* and the generation of the corresponding node in the abstract syntax tree (see Line 4 in the algorithm of Listing 5.2.7). Please see Figure 5.7, showing how token stream subsets $ID\{ID\}$ that reflect definition calls T^*_{call} get mapped to the predefined regular expressions D_{RegEx} generated from the APRIL definitions. Please note that the operation $match$ is similar to the conventional and well-known matching mechanisms for regular expressions.

At this point in the resolution of T^*_{call} from which we can assume that it is of the form of the example statement shown in Listing 5.6, the symbols representing the variables in that statement have to be resolved in a subsequent step. Therefore, it is necessary to restructure T^*_{call} into a set of symbols each applicable to a new embedded definition call and therefore recursively resolvable by the production rule $pathNameOrDefinitionCall'$. Because in that state of the automaton in which the production of $pathNameOrDefinitionCall$ of the level above holds, it is valid to say that $match(D_{RegEx}, T^*_{call})$ holds for resolving any subordinate production rule $pathNameOrDefinitionCall'$ on $T^{*\prime}_{calln}$. Hence, it is possible to introduce a simple elimination operation $\Delta(D_{RegEx}, T^*_{call})$ to yield the set of parameter symbols P^* by eliminating any constant $Const_{D_{RegEx}}$ of the corresponding definition D_{RegEx} in T^*_{call} and keeping the indices of the respective constants found in T^*_{call}. Essentially, the elimination operation Δ works as outlined in Figure 5.8, whereas the detailed mechanism of Δ is presented in algorithm of Listing 5.2.8. Once the positions of the tokens representing an embedded pathname or definition call are determined, it is easy to extract the symbols of the embedded calls $\{T^{*\prime}_{call0}, \ldots, T^{*\prime}_{callN}\} \equiv P^*$.

Therefore, consider Figure 5.6, in which the starting and end tokens are marked (1b(egin) to 3(e)nd) using the corresponding D_{RegEx} of APRIL definition D1. With the help of each marking pair (<n>b to <n>e with n is a natural number), the embedded definition call can be extracted (see also Line 36 of the algorithm in Listing 5.2.8). In the context of the example of Figure 5.6, the TokenStream that represents the definition call

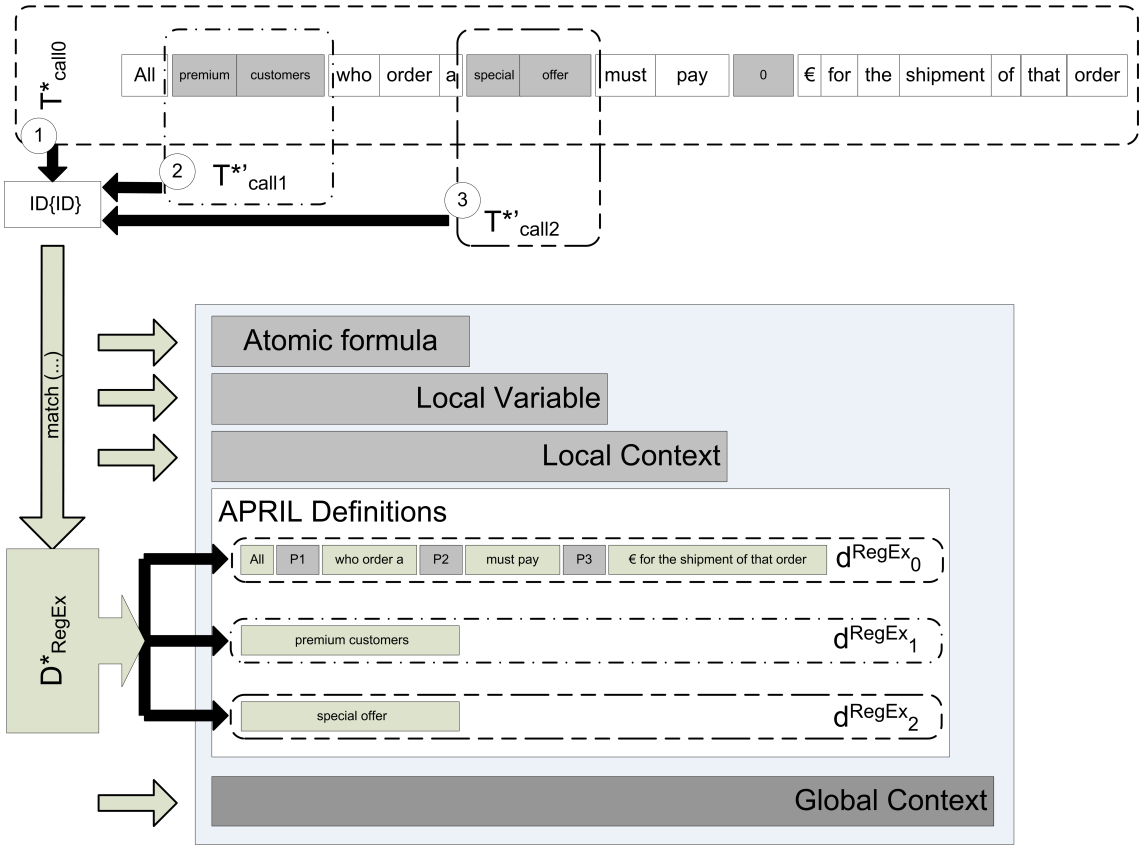


Fig. 5.7: Mixfix parsing schema example

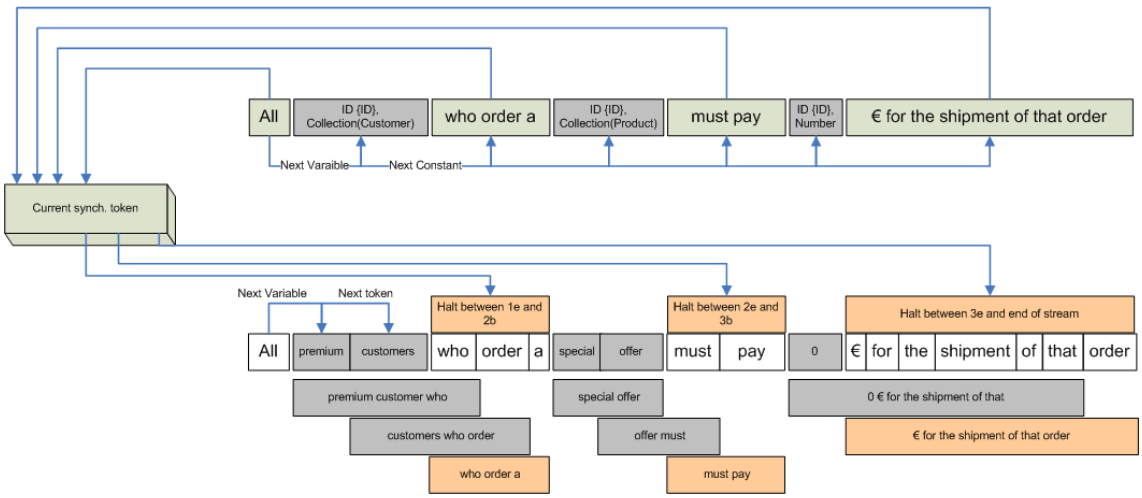


Fig. 5.8: Extracting the embedded variable symbols of a mixfix call (algorithm Δ)

to D2 is represented by the TokenStream ($T^{*\prime}_{call}$) reaching from position $1b + 1$ to $1e - 1$ within T^{*}_{call} of D1. The same holds for the rest, the embedded definition call to D3 and the constant 0. Once all of the sub-TokenStreams for any embedded definition call $T^{*\prime}_{call}$ in T^{*}_{call} are identified, the resolution mechanism traverses downward recursively until every definition call is resolved and represented in the abstract syntax tree.

```
input: string[] tokenStreamOfDefinitionCall, RegularExpressionDefinition D_regex
output: string[string[]] embeddedPathNameOrDefinitionCall
begin
  int tokenIndex := 0
  int definitionCallLenght := number of elements in tokenStreamOfDefinitionCall
  RegexConstant currentSynchToken := get next constant in D_regex
  foreach regexElement in D_RegEx
    increase tokenIndex by 1
    if regexElement is RegexVariable then
      Set StartIndex of regexElement to value of tokenIndex
      int regexElementIndex := 0
      do
        if next constant in D_RegEx is null then
          Set EndIndex of regexElement to value of definitionCallLenght - 1
          exit do
        end if
        string lookAheadString = "";
        for int lookAheadIndex = 0; lookAheadIndex < lenght of currentSynchToken;
 increase lookAheadIndex by 1
          lookAheadString := lookAheadString + element with index (lookAheadIndex
 + regexElementIndex + tokenIndex) in tokenStreamOfDefinitionCall
        end for
        if currentSynchToken is not null and lookAheadString equals
 currentSynchToken then
          tokenIndex := tokenIndex + regexElementIndex - 1
          Set EndIndex of regexElement to value of tokenIndex
        end if
        increase regexElementIndex by 1
      while currentSynchToken is null or lookAheadString not equals
 currentSynchToken
    end if
    if regexElement is RegexConstant then
      if currentSynchToken is not null then
        tokenIndex := tokenIndex + lenghtOf(currentSynchToken) - 1
      end if
      currentSynchToken := get next constant in D_regex
    end if
  next regexElement
  int i := 0
  foreach regexElement in Variables of D_RegEx
    Set element with index (i) in embeddedPathNameOrDefinitionCall := arrayOf (
 tokenStreamOfDefinitionCall from StartIndex of regexElement to EndIndex of
```

```
      regexElement)
38        increase i by 1
39    next
40  end
```

Listing 5.2.8: The operator Δ

5.3 Transformation into the target language

The final step of the transformation process of APRIL into the target languages is the code generation step. Therefore, the finalized abstract syntax tree (AST) (see Section 5.2) is traversed, and according to the AST node that is currently being processed, the corresponding text template for the respective target language operator is selected. Here each node in the abstract syntax tree contains sufficient information about the resolved concept to be mapped to its corresponding counterpart in the target language. The concept of the target language materializes in a template. Section 4.6.7 contains a list of default atomic formula mappings. The template engine used is the ANTLR StringTemplate framework [93]. The basic functionality of the framework is to pick the constituents of an AST node of the parse tree and hand them over to the template in which the framework replaces the placeholders with the actual values contained in the abstract syntax tree node. Figure 5.9 shows how the constituents of the earlier-mentioned *moveto*-operator (see Listing 4.5) are mapped to the placeholders in the template. The substitution is indicated by red, green, and blue coloured symbols *Store*, *Bay*, and *Gate1* picked from the AST node *MOVETO* and passed over to the template. Here the placeholder symbols *source*, *target*, and *routeNode* for the parameters of the StringTemplate interface and the corresponding leaf names of the AST subtree of the *moveto* operator stand for the link between the actual symbols in the statement and the placeholders getting substituted when the template is instantiated. The example code generation step in Figure 5.9 considers the constructed AST node specified in Listing 4.4 and shows how the placeholders in the Template get replaced, yielding the finalized Tempura statements in Listing 4.5.

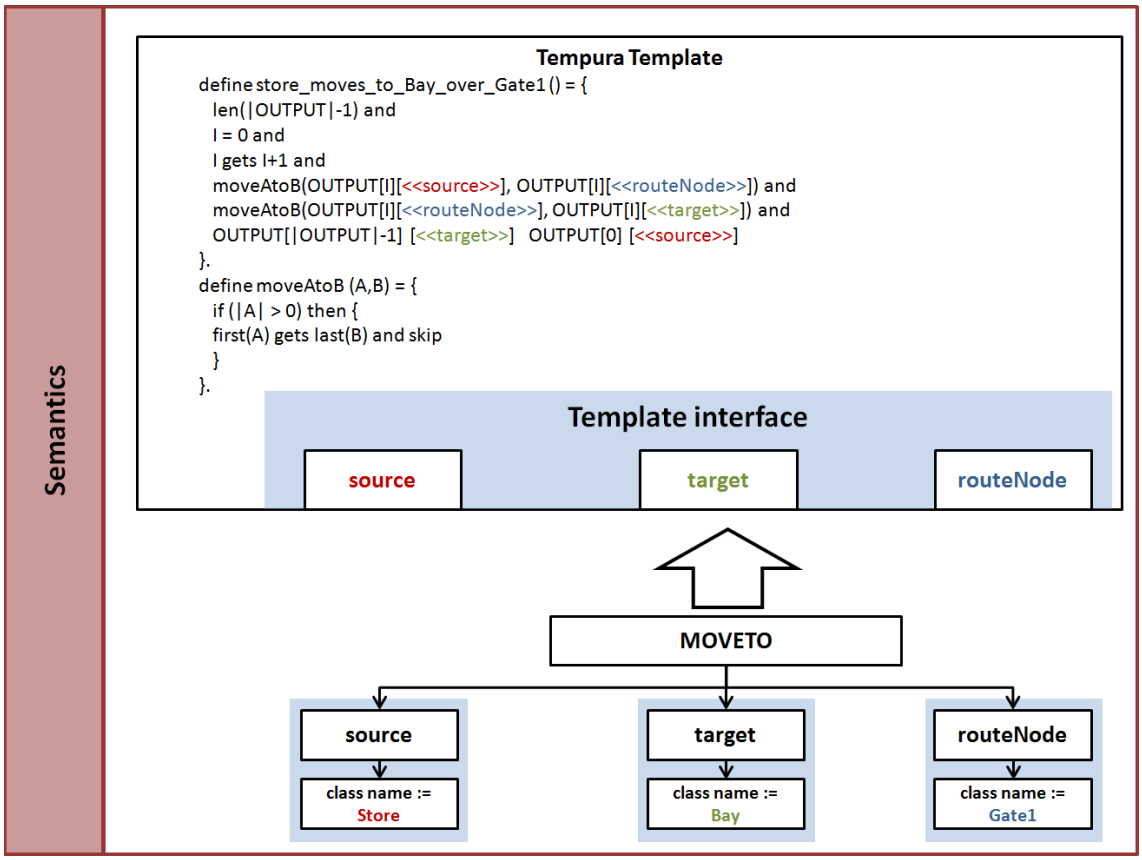


Fig. 5.9: Replacing the placeholders in an ANTLR StringTemplate with the symbols from an AST node

5.4 The APRIL compiler architecture

For the transformation of business rules stated in the APRIL language into the executable target languages, a proper compiler is necessary. Hence, this section aims at outlining the basic (sub)components along with the related data to be processed in order to get from an APRIL rule to a statement that allows asserting whether the predicate is true or false for a given data set. The connection of graphical UML-class model for defining the universe of discourse on which the actual APRIL constraints are based on requires additional connectivity, materializing in additional generation steps that are unlike typical programming languages. This and two additional features, namely the extension of the grammar for incorporating atomic formulas and the specialities of mixfix parsing, make the architecture of the April compiler particularly complicated. These reasons justify the need to provide a detailed insight into how the APRIL compiler works, along with a description of the components and their functionality and interaction. For didactic reasons, a description style based on use cases is chosen, which is supported by a supplementary image to be found in Figure 5.10. Additionally, the image of the architecture (Figure 5.10) is considered for good comprehensibility of the rationale of the components, each in the scope of the overall context.

5.4.1 Typical use cases

The main uses case is the rule implementation. This task is based on a domain model, and as a subsequent step, it typically leads to extracting frequently used constraints that are tailored to the target domain. Therefore, an iterative process of defining, checking, and adapting certain statements in the new APRIL definition has been conducted to make the most out of the new common constraint at that stage. From this position, it is necessary that in the checking phase, the validity of the statements can be tested against a defined test set of test objects, to gain certainty that the new definition works as intended in a productive environment. This is a process common in writing software. However,

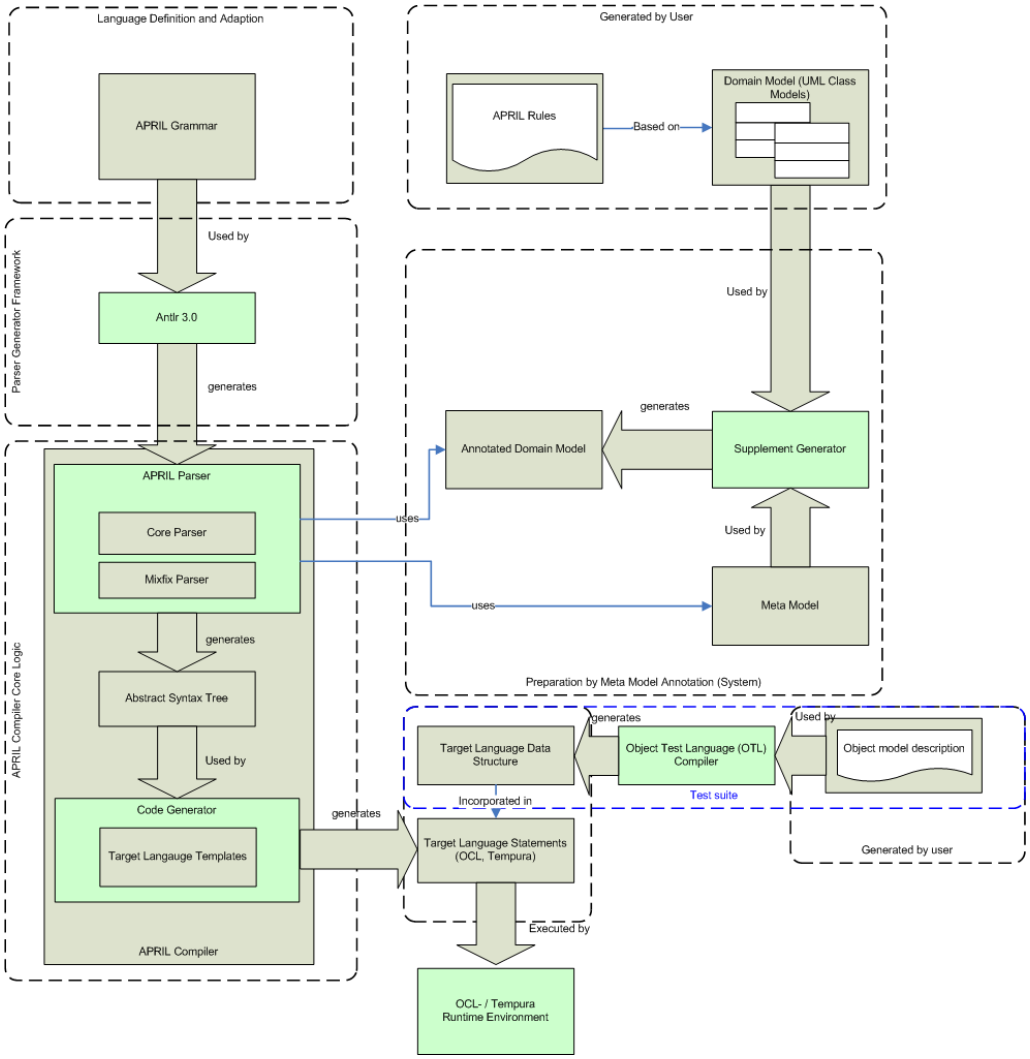


Fig. 5.10: APRIL compiler architecture and related components for handling the main use cases

the difference between typical programming languages and APRIL is that APRIL lacks a proper runtime environment, as it has a denotational semantics. This, next to other reasons, is why the toolchain for executing the compiled APRIL statements is different to mainstream languages and therefore a more detailed description is given in the following subsections.

5.4.1.1 From APRIL rules to target language

It is valid to assume that a business rule formulated in APRIL and its corresponding domain model grow in parallel and are part of the same iteration process that comprises

their definition, check, and adaptation. From the viewpoint of the APRIL parser, the types and symbols used in a statement must correspond to those existing in the current version of the domain model, which is a UML-class model. In order to keep the implementation of the compiler independent from the domain model concepts, contracts based on the metamodel of the used class-modelling notation are used for handling the class model concepts in a generic way. This makes it necessary to produce an annotated class model as a first step. The annotation incorporates the respective metamodel types of, e.g., classes, properties, and associations to be able to traverse the concepts used in the class model in a generic way, which is necessary for decoupling the concrete domain model concepts from the parser logic. Otherwise, it would be necessary to adapt the parser logic corresponding to each change in the domain model. Once the annotated domain model is generated, the APRIL parser works exclusively on the metamodel concepts and treats symbols from the domain model as instances of the metamodel.

The main component of the parser, on the one hand, is the core parser that is directly created by the parser generator ANTLR by transforming the grammar into an automaton. And on the other hand, there is the mixfix parser described in Section 5.2, which implements some very uncommon parser logic that is not unambiguously definable by means of EBNF. Both components core and mixfix parser are necessary to produce the abstract syntax tree, which is an object-based representation of any APRIL statement, enriched with a complete set of information to unambiguously parameterize the target language templates incorporated in the code generator. The code generator traverses the nodes of the abstract syntax tree and fills the placeholders in the target language templates with the symbols of the corresponding operation of the syntax tree node (see also Figure 4.2). Once the code generator has finished and the respective OCL and Tempura statements are generated, the corresponding runtime environments are able to execute the program code, respectively.

5.4.1.2 Validating the test code

As the intention with APRIL is to generate test code, the respective code of the target language is consequently also the test code. However, any code, including test code, has to be validated, which, from the viewpoint of a practitioner, makes it necessary to use a form of test data. The test data are defined as an object model description based on the related class model of the domain with the help of the Object Test Language (OTL) (defined in Appendix C). An OTL description can be translated into a data structure of the respective target language (either Tempura for rules of behaviour) or an appropriate representation of the OCL runtime used (e.g. the Dresden OCL toolkit, the USE Tool, or any other representation). The intention of executing the generated target language code running against a defined set of test data is that a user yields a quick system feedback on the correctness of the defined APRIL statement.

5.4.1.3 Extending the expressive power

As mentioned before, APRIL allows a continuous tailoring to specific domains, which reaches from simply specifying APRIL definitions down to extending the language itself, by incorporating customized atomic formulas. Well-defined atomic formulas provide a significant enhancement to APRIL's expressive power. For the introduction of a new atomic formula, the APRIL grammar has to be extended by typically adding a new production rule that describes the atomic formula to one of the function lexicons, depending on the intended result type. The detailed description of the mechanism was presented in Section 4.2.6. Once the new formula is added to the grammar and the grammar's unambiguousness is guaranteed, ANTLR is used to generate the core part of the APRIL parser.

5.4.2 Summary of the generative components

Table 5.7 outlines the artefacts consumed and generated by the components of the APRIL compiler.

Component	Input Artefact	Output Artefact
Antlr 3.0	April Grammar	Core Part of the April Parser
April Parser	• Annotated domain model • April Rules	Abstract Syntax Tree
Code Generator	Abstract Syntax Tree	OCL and Tempura Statements
Supplement Generator	• Meta Model • UML Class Model	Annotated Domain Model
Object Test Language Compiler	Object model description	Target Language Data Structure
OCL-Runtime	• OCL Statements • Data Structures	Truth table of passed or violated business rules
Tempura	• Tempura Statements • Data Structures	Truth table of passed or violated business rules

Table 5.7: Consumed and generated artefacts

5.5 Summary

This chapter shows the technical aspects of bringing together APRIL statements with its domain model. It presents the processing steps and interaction of the responsive components while APRIL statements are parsed and translated. The core aspect is to show how the resolution mechanisms work considering the typed model concepts used in the statements to unambiguously produce an abstract syntax tree to be used for code generation. Challenges and limitations imposed by mixfix parsing are presented in detail to demonstrate that mixfix definitions can be implemented and used in practice, even if they are nested. The chapter concludes with an overview of the technical components and how they interact in the context of some common use cases occurring in the requirements engineering practice.

Chapter 6

Methodology for using APRIL

6.1 Introduction

Testing is very important for ensuring quality in the software creation process. Therefore, the methodology of applying the APRIL framework for formulating business rules to construct tests for (parts of) a software has been devised. Its intention is to guide the transformation of user-original specifications written in pure, natural language to a formal test specification in APRIL and UML class. The methodology consists of the following steps: The first step has to be done manually, which is to prepare the requirements specification in a way that any functional and also non-functional requirement is refactored to a sufficient degree of detail. Chris Rupp's book [98] contains some very useful instructions to get from a raw and rather coarsely grained specification to detailed and clear requirements reflecting domain properties and its functionality in a degree of detail that a software developer or architect is able to work with. A second step is to extract the system concepts and their interconnections by giving each an exact and unambiguous meaning. If the synonyms are reasonable, this can be made explicit. The result of this extraction step is a business vocabulary that can be easily transformed to a UML-class model, which is the obligatory basis for any subsequent step and for applying the APRIL framework. Once the UML-class model of a domain is elaborated to a satisfying extent, the subsequent step is the definition of formal constraints on the domain model, which best reflects the business rules obligated by the domain logic. This has surfaced to be the most sophisticated step and requires skills in the disciplines of comprehending logic and set theory as well as in specifying formal grammars. A very challenging task in this context is to bring together linguistic considerations with the abilities that a computer-processable language offers, which is in the case of APRIL supported by means of mixfix notation. Typically, APRIL specifications undergo several optimization cycles until an acceptable verbalized version of the formal business rule is developed, which indeed requires a creativity on the part of the requirements engineer. However, once the UML-class model along with the APRIL rules is complete, the steps of test code generation and test execution upon the historical data gathered on the actual

implementation should work automatically as far as possible depending on the degree of integration of the APRIL framework into the utilized software development process. Finally, the last step in dealing with interpreting the test results is performed manually within the APRIL framework. Especially the interpretation of the model checks of the MONA tool introduced by the behavioural business rules requires a deep understanding of the domain model and the possible states that it can be in. However, interpreting formal model check results is not covered in this work and can be delegated to future work.

Please consider Figure 6.1 for a brief overview of the concrete steps that the methodology recommends. Here, each arrow of the arrows in the process description stands for a concrete step in the methodology sketched above. The *formal verification* step is considered as well and can be examined in detail in Chapter 7.

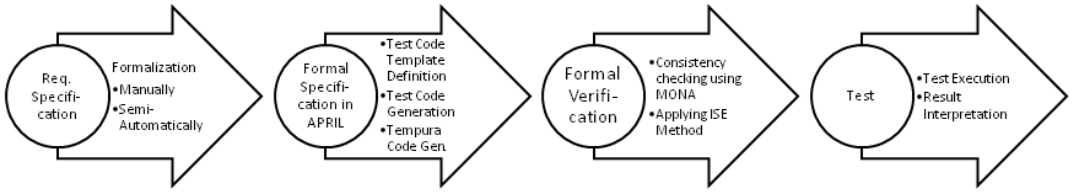


Fig. 6.1: Adaptation of the basic requirements engineering methodology

The following section describes the context of the APRIL framework within software development processes based on testing and the scope the APRIL language addresses to constitute an advance in these processes.

6.2 The APRIL framework in the context of software development

Modern software development processes for large-scale software strictly require incorporating tests to keep the overall quality of the implementation at a defined level. Independent of the development process chosen (e.g., agile processes, waterfall, etc.), testing takes an important role in software development as it helps to detect errors in an early stage of the software creation process. It has been proved for many decades that the later an error found, the more expensive to fix it. Based on that, it has become common practice to use tests as a manifestation of the artefact (productive code, test) and actor (developer, requirements engineer) diversity in development processes. Diversity like this has been made explicit in modern software development processes. Typically, these processes dedicate two diverse branches, both sharing a user-original specification document. One branch is the actual implementation conducted by developers, whereas the sibling branch is conducted by different actors, constantly testing that the productive code artefacts created do work as specified (see Figure 6.2). According to the different stages in the process, three characteristic types of tests have crystallized to be reasonable at each corresponding stage of development. These three types are module-, integration-, and system testing, each targeting a certain scope, beginning from a coarsely grained and from a functional viewpoint fully domain-related scope at the system test level, down to the integration test level which can involve technical aspects of the implementation, and finally, the module or component test to verify a functionality at the finest-grained level. The importance of testing involving all the three test types is even emphasized when development processes are chosen that pursue a strict incremental approach. The reason is that tests are usually automated and can thus be utilized for regression testing, allowing to verify the correctness of the already implemented functionality. Approaches using automated tests raise the efficiency of software development [8].

As mentioned earlier, the APRIL framework targets avoiding discrepancies between

the formalized, automated tests and their corresponding natural language counterparts in the specification document. Please consider Figure 6.2 which shows how APRIL integrates into a software development process that significantly considers testing[1]. The relevant components of the framework are shown in Figure 6.3. A core engine containing an editor and a translation engine helps to produce Tempura/OCL statements from business rules based on a corresponding UML-class model. The interesting thing with consistency checking (described in detail in Section 7.4) is that a toolchain with PITL2MONA and MONA is used to allow analysing the ITL-based Tempura statements produced. Based on the results of the consistency check, a refactoring of the behavioural business rule itself or the used Tempura template can be reasonable.

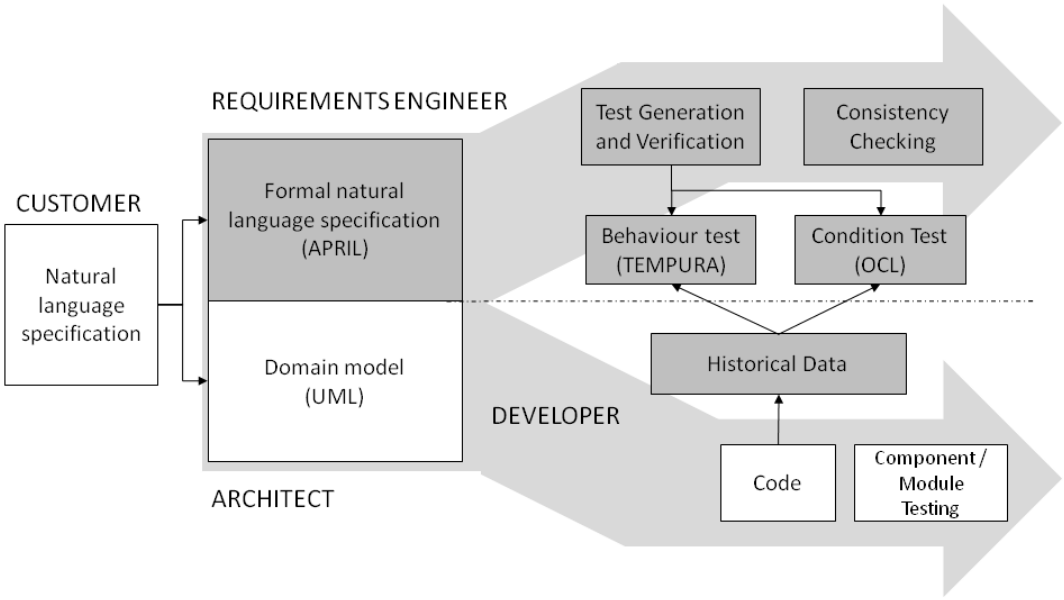


Fig. 6.2: The APRIL framework in the scope of a development process involving tests

Consistency checks are possible because APRIL is based on predicate and temporal logic. As an example for applying consistency checks, Section 7.4 shows how reasoning on behavioural rules (based on Tempura) can be integrated into the APRIL framework.

[1] which also applies to processes applying the widely used V-Model

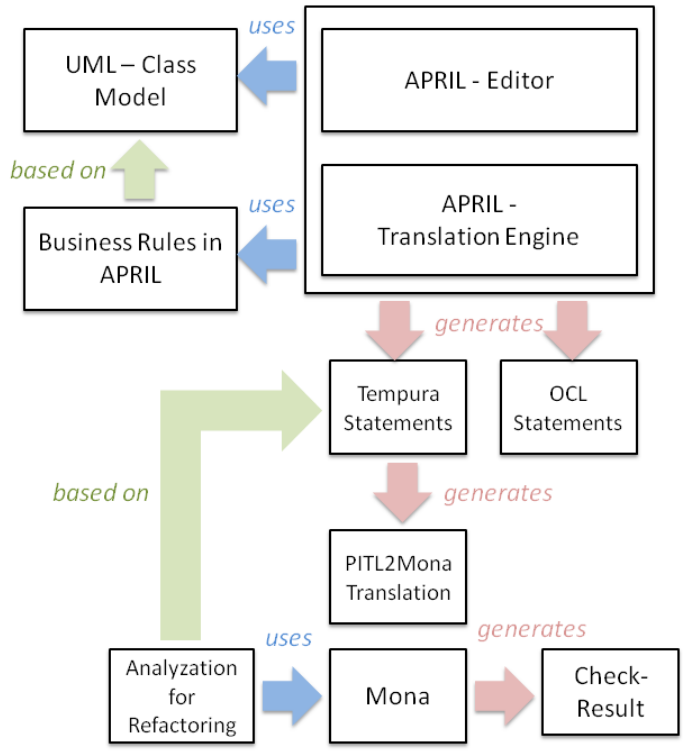


Fig. 6.3: Overview of the components of the APRIL framework

On the one hand, the APRIL framework can support incremental development processes for conducting such consistency checks to initially verify that formalized requirements (APRIL rules) are free of contradictions. On the other hand, APRIL's main task is to support the generation of test code to verify that the implementation complies with these formalized requirements. To help in formalizing the requirements documents, a methodology has been developed, which is presented in Section 6.3 and can be seen as a guideline to transform relevant parts of pure natural language specifications into a set of APRIL rules.

6.3 Enhancing raw requirements documents with APRIL

It is often difficult to transform raw requirements specification documents directly into UML-class diagrams and formal business rules as their quality is in practice very heterogeneous. Hence, initial tasks for overcoming the problems imposed by bad

requirements specifications are left to the human user rather than nlp-based automatisms. Human creativity should (still) be superior to the abilities of an nlp automaton when it comes to creating formal statements or templates of statements that sound like natural language in the context of APRIL. APRIL only takes the helping part that allows annotating the semantics behind these statements and applying automated checking functionality to help avoid errors in the specification. An example of such checking functionality is to find trivial errors such as type mismatches of, e.g., misuse of certain parameters in the wrong context. A guideline on how to arrive at APRIL statements from a natural language business rule is sketched out in the Sections 6.3.1, 6.3.2, 6.3.3, and 6.3.4. The application of the phases showed that each phase cannot be seen as a single, self-contained step after which the result is completely finished serving as the input for the next phase. The presented phase sequence is only a practice-based manual to guide an iterative refactoring of a raw version of a specification into a set of APRIL rules.

6.3.1 Pre-preparation of the requirements document

Before the transformation from pure natural language requirements into APRIL statements can be started, the requirements document has to be refactored into a stage where all requirements are made explicit and no implicitly hidden requirements are left. Chris Rupp (see [99] pages 134-163) describes some very useful steps and strategies to raise the quality level of a requirements document, which is considered to be adequate for the subsequent step (Section 6.3.2). Since [99] is only available in German, the Table 6.1 of requirements transformation rules has been extracted and translated.

Moreover, the presented refactoring rules of Table 6.1 help to maintain a certain degree of quality for requirements formulated in pure natural language as well as APRIL statements. The quality criteria are also described by Rupp [99] (page 26) and can be subsumed as presented in Table 6.2.

Resolve nominalizations and add a new requirement for each detected nominalization
Refine verb phrases in requirements by the following aspects: • What does the verb describe? Make sure that the verb is unambiguously used in the context of the requirement and it is underpinned with a precise meaning • What is the subject in the verb phrase? Make sure that the subject has a precise meaning. • What is the object in the verb phrase? Make sure that the object has a precise meaning. • Are there certain adverbs raising the degree of detail that the verb phrase is describing? If so, give them an exact semantics. • Are there any preconditions for applying the requirement containing the verb phrase? If not, extract them and make them explicit, e.g., in an additional requirement. • Is the verb phrase, the subject, and the object correctly quantified?
Refine imprecise substantives
Requirements with an incomplete precondition structure should be checked and formulated in detail or be replaced by a new requirement
For each unverbalized, implicit assumption, one or more additional requirements have to be created
Express processes by unambiguous main verbs.
For each main verb, use exactly one requirement statement.
Refine and complete missing information to the properties described.
Terms describing the properties of concepts must be measurable and testable.
Factor non-functional aspects out of the dedicated requirements.
Remove any clauses that do not matter from the context of the requirement.

Table 6.1: Requirements for improving rules relevant to APRIL, according to Rupp [99]

Quality Criterion	Description	APRIL enabler
Complete	A requirement must contain any information that is necessary to describe a system's features. This means that the described features must be testable.	APRIL is supposed to be transformed into test code
Necessary	A requirement should only describe a service, a property, an edge condition, or a quality feature that is really needed to fulfil an aim in the context of the system to be described	There is only one obligation level in APRIL, namely, *must*; no *can* and no *should*.
Atomic	One requirement statement contains exactly one requirement with a single process verb.	Decomposition
Traceable	Each requirement has to be traceable back to its source, e.g., a specific statement of a stakeholder or an abstract requirement that it has been derived from.	Naming of rules
Consistent	All requirements have to be free of contradictions	Partly possible: consistency checking using model checkers (e.g., MONA) can be done. Wrongly specified but consistent statements cannot be detected.
Unambiguous	A requirement must be clear and can only be interpreted in a unique way	Unambiguous interpretation is ensured by the APRIL parser/compiler. The natural language part is the responsibility of the user.
Testable / Measurable	The described feature has to be testable or measurable	Basic intention of APRIL

Table 6.2: Quality criteria of requirements according to Rupp [99]

6.3.2 Extracting business concepts for the domain model

In Phase one, the natural language specification is refactored to reduce ambiguities and extract business concepts that can serve as domain model concepts materializing in UML classes. Therefore, relevant domain concepts have to be identified and collected, e.g., in a terminology list. Typically noun phrases in the role of the subject and object in a sentence constitute domain concepts. In this context, the verb phrase in the same sentence indicates the type of relationship between the two concepts and hence, the verb phrase should be considered for building the conceptual domain model as well. The resulting artefact of the intermediate collection step contains potential class names (as domain concepts) as well as their roles if two or more concepts are related that can be taken as role names of UML associations. This collection serves as input for the following step that is to define the exact meaning of each element in the preliminary list of domain concepts, which should consider the context of the entire domain in order to avoid ambiguities. The aim of the next step is to condense the preliminary list, by eliminating any ambiguities left. One output artefact of this phase is, next to others, a *refined requirements specification* and the first version of a *domain model*.

Here are some useful guidelines for the refinement step:

- Eliminating redundant synonyms helps to simplify business terminology and reduces the danger of creating ambiguous business rules.
- The use of pragmatic noun phrases should be avoided, which means that, e.g., substantives that impose a different meaning in a subordinate scope should be redefined, or at least be explicitly named, using a dedicated domain concept that allows concluding the context it is used in.

Once each domain concept represents a unique meaning, it is possible to form rudimentary business rules that are more formal than the original business rules. This can be achieved by projecting the concepts back to the meaning of the original business rule context and assigning them to the properties obtained from the contexts of the business

rules. A property of this form is typically either a predicate or a set comprehension. An annotated domain concept using a property of the aforementioned form can be seen as a derived concept. Any concept, if derived or not, can be related to other concepts. Remember, in natural language, relations can be expressed by verb phrases glueing the noun phrases in the roles of subject and object together, which is where the transformation should finally lead back to. But for this first step, is it sufficient to gain an overview and understand each domain concept under consideration. Comprehending large and complex business rules involve many domain concepts; it is, therefore, reasonable to temporarily leave the conventional textual representation notations and use a graphical format, which, for example, can be a mind map. However, at this point, anything that helps in comprehending the meaning of the business rule to be formalized should be allowed. The aim of this step is an intermediate representation of the specification with ideally no ambiguities and a well-defined business vocabulary, for which is important for stakeholders to have a clear universe of discourse that helps minimize misunderstandings. For later reference, the resulting artefact of this stage shall be referred to as *mind map model*.

Approaching the initial formalization of business rules in the way presented can help reveal inconsistencies and conflicts in a very early stage as practice shows that this phase produces the deepest understanding of the business rules and its domain concepts.

6.3.3 Abstracting and defining reusable sentence patterns

The *mind map model* defined in the previous phase is used to find textual specification patterns for similar classes of specifications. This can be done by freely defining properties and relationships of business concepts in a natural language phrase or in parts of a phrase in which linguistic noun phrases can be replaced by typed placeholders. For placeholders, class names can be used as type symbols. The defined sentence patterns are meant for transformation into APRIL definition signatures. Bearing in mind that the more general the signature definition is, the higher is the degree of reuse in the later

specification. However, the practical application of this approach surfaced in that the more generic the pattern is supposed to be, the more difficult it is to find a suitable natural language representation that can fit in a grammatically correct way into the sentences it shall be used in.

For building reusable sentence patterns as APRIL definition signatures, consider the following guidelines:

- Properties of concepts are represented by linguistic adjectives or short relative clauses.
- Linguistic verb phrases shall not imply new concepts, properties, or relations. If this is not avoidable, the new verb phrase should be added to the *mind map model* as it can also imply a new association between the corresponding UML classes represented by the related noun phrases.

The result of this phase should be a preliminary set of *reusable sentence patterns* that can be used as APRIL definition signatures. In practice, this step is likely to be passed through multiple times (see Figure 6.4), hence it has shown that the context in which the patterns are used can retroact heavily on the morphology of the pattern. As mentioned before, balancing the semantics based on logic and the linguistic representation of general patterns in a way that the instantiated pattern constitutes a clear and naturally sounding sentence, requires considerable creativity, which is an ability exclusive to humans.

6.3.4 Formulating APRIL statements

This phase requires the artefacts created in the preceding phases, which comprise a *mind map model* that structures an example domain model, a *refined requirements specification* document that is now ideally written in a rigorous technical style, and the set of *reusable sentence patterns*. The aim is to reformulate the requirements of the *refined requirements specification* using as many *reusable sentence patterns* parameterized with concepts that are suitable for the requirement. Practice shows that this is reasonable in the majority

of the cases, especially if the pattern is supposed to be used directly subordinate to the APRIL rule. For choosing the correct semantics based on APRIL atomic formulas, the *mind map model* can help to outline the meaning behind the *reusable sentence patterns* used.

Basically, this phase can be subdivided into the following substeps:

- Decomposition. Separation of the content of the business rules by considering reusable sentence patterns (APRIL definitions in mixfix notation) or formulating the meaning of the business rule directly as atomic formulas. A further decomposition of the logic statements can also be supported by using local variables or local methods. The result of this step is mainly influenced by keeping the compliance of APRIL statements with natural language as good as possible.

- Formalization. The logical essence of the definition is formulated in the body of an APRIL rule or definition using atomic formulas in which the common constraints subset can be very helpful in keeping the statements concise. The formalization of an APRIL business rule is more a matter of understanding logic and set theory than applying tricky linguistic schemes.

- Extension. In some cases, it may be helpful to extend APRIL's set of atomic formulas, e.g., when a frequently occurring domain-specific term can make business rules a lot more intuitive.

The necessity of forming new sentence patterns may be indicated by the following:

1. The *mind map model* describes a set comprehension on a class of domain concepts.

2. The meaning of a relationship between two concrete domain concepts can be generalized to a relationship of the corresponding classes.

In those cases (1 and 2 from enumeration above), it may be reasonable to define a new sentence pattern that can be added to the list of *reusable sentence patterns*.

If an abstraction makes sense, the following steps can be undertaken to add that newly defined property to the universe of discourse:

1. Decide which logic predicates can be clustered with respect to the *mind map model* and the *domain model*. Predicates are indicated by properties in the *mind map model*.

2. Decide which clauses can simply be cast into local variables, e.g., set-operations or logic expressions.

3. Decide which clauses from the *refined requirements specification* can be reused to form an APRIL definition with respect to reusability, and think of an appropriate signature, potentially utilizing the mixfix notation.

4. Definition composition and signature readjustment loops (see the previous step in Section 6.3.3). This step includes the fine-tuning of the APRIL statements (e.g., mixfix definition expressions) in terms of an optimized natural language sentence.

It should not be surprising that in this phase it may arise that certain model concepts have to be adapted for a better natural language reading of the final APRIL business rule. The practical experience with APRIL shows that a sporadic refactoring can be necessary for fine-tuning. Especially the following aspects of APRIL statements as well as the UML model should be mentioned:

- adapting names of UML-class roles or, in certain cases, their cardinality if it appears suitable in the modelling context.

- adding new associations to introduce concepts (as roles) to better support the natural language reading of the statements they are used in.

- adding/removing methods to a UML class (same as with the associations)

- adding attributes to UML classes as auxiliary state variables and also to introduce a concept to improve the natural language reading in which it is used in.

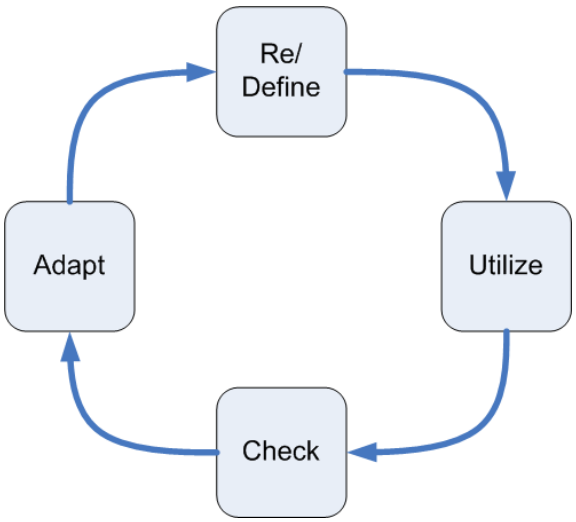


Fig. 6.4: Elaboration cycle of APRIL statements

- changing the mixfix patterns to make them suitable to natural language grammar rules.

Eventually, this phase yields the transformed specification into the APRIL language.

6.3.5 Working with Tests generated from APRIL statements

Once the APRIL statements are defined the transformation into the respective target languages can be done automatically. Here APRIL statements declared with the keyword *Behaviour* are translated to Tempura statements and all other statement typs not introduced with the keyword *Behaviour* are translated to OCL. The transformation of APRIL statements into its target languages is described in the Chapter 5.

For executing OCL and Tempura statements it is necessary to supply historical data recorded in the productive system. As mentioned in Chapter 5 this can be done by using Aspect Oriented Programming mechanisms e.g. by defining pointcut filter statements for each getter- and setter- method that represents a domain model concept implemented in the productive implementation. However, for the sake of keeping the example as comprehensible as possible we use direct tracing calls in the example java code. The example code is in Listing 6.3.1 and shows the implementation of a UML-class named *CarDoorController* having the two attributes *Vehicle_Locked* and *BatteryVoltage*. Each access of an attribute via its getter- or setter-method contains a line of code that prints the value of the accessed attribute in a history that can be saved in a file. Please note that the technical details of the logging mechanism is omitted here. However, even if the exact formatting depends on the concrete implementation and the preferences of the used runtime, Table 6.3 shows an illustrative example of history data, that is based on Listing 6.3.1.

Each time a concept of the domain model is accessed (e.g. a class attribute here) the historical data is extended by the given value that is supposed to occur for the intended test scenario. Therefore, the productive implementation has to behave according to a defined scenario that fits an APRIL rule, that is defined to meet that test case.

```
1 public class CarDoorController {
2     bool Vehicle_Locked ;
3     int BatteryVoltage;
```

```
4
5     bool Get_Vehicle_Locked() {
6        return Vehicle_Locked;
7        AppendToHistoryInFile(Vehicle_Locked);
8     }
9
10    void Set_Vehicle_Locked(bool vehicle_Locked) {
11       Vehicle_Locked = vehicle_Locked;
12       AppendToHistoryInFile(Vehicle_Locked);
13    }
14
15    int Get_BatteryVoltage(){
16       return BatteryVoltage;
17       AppendToHistoryInFile(BatteryVoltage);
18    }
19
20    void Set_BatteryVoltage(int batteryVoltage){
21       BatteryVoltage = batteryVoltage;
22       AppendToHistoryInFile(BatteryVoltage);
23    }
24
25    void AppendToHistoryInFile(int value) {...}
26    void AppendToHistoryInFile(bool value) {...}
27    void AppendToHistoryInFile(...) {...}
28 }
```

Listing 6.3.1: Java class example

"Vehicle_Locked"	"BatteryVoltage"
true	12
true	9
false	8
false	8

Table 6.3: Example History Data Based on Listing 6.3.1

An APRIL business rule shown in Listing 6.3.2 evaluated over the historical data of Table 6.3 holds, respectively. Please also consider the Tempura translation in Listing 6.3.3. The natural language rule reflecting the intention of the afore mentioned formal APRIL rule in Listing 6.3.2 may state that: *"For safety reasons it must be guaranteed that*

if the electrical energy source cannot power the door locks allowing the occupant to leave the car in an emergency, the door locks must be released."

```
1 Behaviour OpenDoorOnLowBattery concerning CarDoorController:
2 it is always that case that (BatteryVoltage < 9 implies that it is not the case that
      Vehicle_Locked).
```

Listing 6.3.2: Example APRIL business rule holding on example data of Table 6.3

```
1 define OpenDoorOnLowBattery() = {
2 always ((BatteryVoltage < 9) implies (~Vehicle_Locked))
3 }.
```

Listing 6.3.3: Tempura Version of 6.3.2

In cases where the business rules are not so simple as in the earlier example, this methodology offers a way to simplify Tempura statements, should its APRIL representation be considered too complex and incomprehensible to human readers. For the application of the simplification, we use the ISE subset of Moszkowski's ISEPI method presented in Chapter 7. The result of the simplification, which is a simplified Tempura formula, has to be done manually. An automated backward transformation has not yet been devised within this work. However, the one-to-one relation of APRIL atomic formulas to Tempura statements does not prevent an automated transformation form Tempura to APRIL.

6.4 Limitations of APRIL

APRIL is designed to concisely express constraints on a domain model in a short, declarative way. It is not suitable to define algorithms that are close to imperative

implementation patterns. The negative effects on the comprehensibility of single APRIL expressions are demonstrated in the "Shift Planning" case study shown in Appendix D. The study shows how complex and non-intuitive an APRIL statement can become when, on the one hand, the specification of a business rule is too large and complex and even decomposition cannot help to overcome this. This is also the case when, on the other hand, the statements get too close to technical implementation details that typically require being expressed imperatively, which even enhance the unnatural impression of the (sub)expressions. The case study also shows that describing complex data structures in APRIL can also lead to awkward statements that are difficult to comprehend. In the context of APRIL, describing data structures should be left to UML-class diagrams.

Another aspect where APRIL suffers certain drawbacks is also in the field of technical descriptions, which is to describe state machines. This is actually not really a hard limitation of the quality that APRIL is not expressive enough, but the description of state machines in APRIL can disturb the readability of a business rule. An impression is given in the "train trip"-case study, where synthetic states where necessary to distinguish in which context the rule is supposed to hold. However, it can be argued that a non-disturbing description of state machines by APRIL rules can be issued to a tricky modelling of the UML-class model in combination with a suitable natural language sentence pattern.

Additionally, aspects concerning the performance of processing APRIL statements are neither considered in the language design (syntax, semantics) nor in the choice of the toolchain of the APRIL framework. The reason is as mentioned, that APRIL works as a means to define test code that is able to work asynchronously on historical data without any interaction with the productive implementation at runtime, which does not impose the need to give performance aspects too much space.

6.5 Summary

This chapter comprises a guideline for transforming purely unstructured natural language requirements specifications into a set of appropriate APRIL statements along with a suitable corresponding UML-class model. Therefore, development process steps are also considered at a very coarse-grained level. The important role of testing in software development and the support that APRIL can provide in this context is emphasized. Therefore, APRIL statements are evaluated against the historical data records gathered from defined test scenarios the productive code is executed against. Finally, some limitations with respect to the natural language formulation abilities of APRIL are given. These limitations mark a not-so-clear border between gaining efficiency by the usage of APRIL and losing efficiency when using APRIL inappropriately, e.g., on complex data structures.

Chapter 7

Advanced methods in requirements engineering

7.1 Introduction

This chapter addresses the detection of contradictions and the simplification of complicated business rule specifications. Therefore, the challenge handled in this chapter is twofold. On the one hand, the presented material shows how to detect contradictions that cause inconsistencies in the business rules. On the other hand, a subset of steps introduced by a novel method called ISEPI (see [77] and Section 7.5) is utilized to abstract very detailed behavioural business rules and thus make them more concise and comprehensive. An additional aim of using ISEPI is to evaluate whether this very new method can be used in real-world business rule specifications and contribute to their improvement. Therefore, Section 7.6 presents the validation of the method using an industrial example specification devised by the *Fraunhofer Institute* in cooperation with the *Daimler AG* [46]. Consequently, the toolchain and ISEPI itself are integrated into the APRIL framework, which is then able to systematically refactor complex behavioural business rule statements. Moszkowski's novel ISEPI is presented in Section 7.5, demonstrating the stepwise reduction of the complexity of a Propositional Interval Temporal Logic (PITL) based statement. It also shows how to detect the necessary preconditions for the application of ISEPI, which complies with (parts of) the business statement with a logical property [76], called 2-to-1 property. This technique is based on a novel axiomatic system within the PITL formalism which is a subset of ITL. The tools utilized for consistency checking and refactoring are MONA version 1.4 [55] in combination with a translator (PITL2MONA [39]) which converts Propositional Interval Temporal Logic statements into MONA-interpretable syntax. The contributions of this chapter can be subsumed by the following two points:

- Detecting contradictions in behavioural business rules.
- Enhancement of the APRIL framework with a stepwise method to reduce the complexity of behavioural business rules using formal reasoning.
- Evaluation of the practical applicability of

Moszkowski's simplification method for ITL statements.

7.2 Propositional Interval Temporal Logic

PITL stands for Propositional Interval Temporal Logic and is a subset of ITL. Most of the concepts of PITL are expressible in Tempura. Time within PITL is modelled by discrete, linear intervals σ being σ_0 for an empty interval or $\sigma_0 \dots \sigma_{|\sigma|}$ for intervals with length $|\sigma|$. Each interval σ_i maps a propositional variable *p* to either *true* or *false*. A subinterval of σ is a sequence of states within σ, such as $\sigma_k \dots \sigma_m$, with k, m as natural numbers and $1 \leq k < m \leq |\sigma|$. This includes σ itself. Infinite intervals are also possible, but they are not utilized here. The following BNF rule for PITL formulas *A* may serve as a reference to the reader, with *p* as any propositional variable, *skip*, "A;A", and "A*" as ITL operators denoting an interval of length two, and *chop* (";") to concatenate two intervals and *chop-star* ("*").

$$A ::= true \mid p \mid \neg A \mid A \vee A \mid skip \mid A; A \mid A^* \ .$$

The operators $\neg$ and $\vee$ have the usual semantics. The constant *true* holds for any interval. *Skip* states that the interval has the length one. $\sigma \vDash A^*$ holds if σ has only one state, or σ is finite and satisfies A, or can be split into finite-length subintervals (even with different lengths) each satisfying A. This can be extended canonically to infinite intervals $|\sigma| = \infty$. *Chop-star* is for intervals what the Kleene-Star operator is for cardinalities of (non)terminals on the right side of the production rules in a formal syntax. The meaning of *chop* was described in 2.6.3. More on PITL and its semantics can be found in [78, 76]. The remaining propositional operators and PITL operators can be deduced from the ones already presented. For a complete representation of the PITL operators, the interested reader may consider [77] which also includes a list of deductions. Any deduced operator that is used later and is important for comprehending the presented example materials, is explained in its local context.

7.3 The 2-to-1 Property

Any PITL formula A is **2-to-1**, for which the implication $(A; A) \rightarrow A$ is valid [77]. 2-to-1 means that if two semantically equal PITL formulas are concatenated with chop, each of the formulas holds on the entire interval. The class of formulas complying with the 2-to-1 property is closed under conjunction and the temporal operator $\Box$ (*always*). Such closure properties help to construct formulas from simple building blocks, guaranteeing that the resulting formula preserves the property of that class, e.g., $(\Box A\,;\, \Box A) \rightarrow \Box A$ and $((A \wedge B); (A \wedge B)) \rightarrow (A \wedge B)$ are also valid. Some examples of 2-to-1 formulas are: true, any propositional variable p, empty, and B where $B \equiv (B' \wedge \neg empty)$, and so are the PITL formulas $\Box p$ and the liveness property $\Box(p \rightarrow \Diamond q)$[1], but not *skip*. *Skip* tests the interval length to be one. However, $skip\ ;\ skip$ states that the entire interval has length two. Please remember that ";" (*chop*) requires the prefix and suffix subinterval to share the adjacent state. Therefore, $skip\ ;\ skip \rightarrow skip$ is invalid. In [77], Moszkowski presents a complete disquisition of the 2-to-1 property, its fundamental rationale, and a complete set of related proofs. This work focuses on utilizing 2-to-1 properties to simplify formulas that represent behavioural business rules (see Section 7.5). Ideally, 2-to-1 can be used to identify invariants that are unintentionally modelled as behaviour, which tremendously contributes to the simplification of business rule specifications.

7.4 Consistency checking for behavioural business rules

It is a known fact that the later an error is found in the system creation process, the more expensive it is to fix. Hence, it is most desirable to detect errors as early as possible. Many errors are inconsistencies imposed by simple logical contradictions which can be found by model checkers, by inspecting the set of business rules, and applying logic inference mechanisms (e.g., see Section 7.5), without knowing anything about the superordinated

[1]The liveness property states that it is always the case ($\Box$) that if a certain condition is met (p) a subsequent reaction has to follow ($\rightarrow \Diamond q$)

domain model or compliance rules [4]. On the one hand, there are quantitative reasons for inconsistencies, which are not only those caused by the pure amount of business rules but also those caused by the evolution of the processes that business rules are specified for. On the other hand, qualitative reasons, attributable to the complexity and high degree of interrelation between business rules, produce inconsistencies that are even more difficult to detect. Both classes can be tackled by a proper tool support, allowing detecting and avoiding inconsistencies [4]. In the following, three types of interesting inconsistencies are exemplified.

- **Partial contradictions**: indicated by the results of a checker stating that a conjunction of certain business rules yields at least one counterexample. For example, an artificial use case from the manufacturing industry may state that a machine has to meet criteria of certain properties while a service action is taking place. The business rules describing those properties are in Listing 7.2, and their conjunctive normal form in propositional temporal logic is in Listing 7.4. The abbreviations of the propositional variables are listed in Listing 7.1. MONA evaluates the formula in normal form in Listing 7.4 and yields a counterexample for the case if the machine is not serviced ($\neg service$) and not in run mode ($\neg run$). In that case, the implication $Prop_3$ is $false$, which in conjunction with $Prop_1$, and $Prop_2$ means that the entire formula is $false$. This means that for the case $Prop_3 \equiv false$, there exists at least one set of states in which the rules defined for the machine is not met. However, once a partial contradiction is detected, it has to be decided whether this is a critical case or not, and whether the rule (formula) has to be extended to remedy exactly this case. Without domain knowledge, a checker cannot know that in this domain-specific use case, it does not matter that a service does not take place when the machine is not running. That is why the interpretation of a checker's result is important, and a considerable amount of domain knowledge and careful investigation is necessary.

- **Contradictions**: indicated by the results of a checker stating that a business rule

door_open	the security door is open
service	the service selector switch of the machine is active
run	the machine is in run mode

Listing 7.1: The abbreviations used in the formula of Listing 7.4

$Rule_1$	If the machine is serviced, the service personnel has to access the working area through the safety door.
$Rule_2$	While the machine is running, the safety door must never be opened.
$Rule_3$	'On the fly' servicing must be done while the machine is in run mode. (E.g., because the construction process the machine is supporting makes this necessary.)

Listing 7.2: Business rules for the behaviour of a machine

$Prop_1$	it is always the case that if the service selector switch is active, the security door is opened subsequently.
$Prop_2$	it is always the case that if the machine is in run mode, the security door must never be open.
$Prop_3$	it is sometimes the case that the machine is serviced while it is running.

Listing 7.3: Meaning of the propositions in Listing 7.2

$$\Box(\underbrace{(\Box(service \rightarrow \Diamond door\_open))}_{Prop_1} \wedge \underbrace{(\Box(run \rightarrow (\neg door\_open)))}_{Prop_2} \wedge \underbrace{(\Diamond(run \rightarrow service))}_{Prop_3})$$

Listing 7.4: Conjunctive normal form of the propositions in Listing 7.3

is cannot be satisfied. If not elaborated with care, adding an additional business rule may cause contradictions. E.g., if the aforementioned behavioural rules 1-3 of Listing 7.2 are taken into account and a fourth one (4. in Listing 7.5) is added that can be formalized as $\Box q$, and a contradiction may remain undetected, then the entire formula consisting of $Prop_{1-4}$ can be formalized as in Listing 7.6. MONA's evaluation result says that this formula cannot be satisfied. The reason for this inconsistency may be obvious in this formalized case, as the business rule is concisely written down in one line. So, it is easy to see that $Prop_4$ now requires run to be $true$ all the time, which implies that $service$ and $door\_open$ get $true$ ($Prop_1$ and $Prop_3$), which is in contrast to $Prop_2$ stating that it must always be that if

$door\_open$ is $true$, run must be $false$. However, in real world specifications, business rules tend not to be presented as concisely as in Listing 7.6, which makes it very difficult for requirements engineers to detect such inconsistencies ranging over two or more implication steps.

$Prop_4$	The machine never leaves the run mode (e.g., for avoiding a time consuming reset before startup, which operating personnel had to face in the past after collecting experience running the machine.

Listing 7.5: Business rules for the behaviour of a machine add-on of Listing 7.2

$$\Box((\Box(service \rightarrow \Diamond door\_open)) \wedge (\Box(run \rightarrow (\neg door\_open))) \\ \wedge(\Diamond(run \rightarrow service)) \wedge \underbrace{\Box run}_{Prop_4})$$

Listing 7.6: Accidentally invalid extension of Listing 7.4

- **Redundancies**: tautologies or if two formulas are semantically equivalent, each of which is generated from a different business rule. This is not really an inconsistency in the sense of an error or contradiction in a business rule's definition, but it impacts a system's performance. The classical redundancy checks for behavioural business rules will not be considered any deeper. However, there is an interesting class of redundancies that comply with the 2-to-1 property. In Listing 7.7, an example for 2-to-1 is sketched, providing an impression of such a non-trivial ITL-based simplification. The Listing shows a formula of the form $\Box(a \wedge \Diamond b)$, which means that it is always the case that if a holds, b holds even several times. In the special context of the business rule the formula $\Box(service \wedge \Diamond(run \rightarrow door\_open))$ states that in the service mode it is possible that in the run-mode, the door can be opened (and closed) several times, typical for a use case in which service personnel accesses the machine.

  The perspectives of this implication are quite intriguing, which is shown in Section 7.5.

$$\underbrace{(\underbrace{\Box(service \wedge \Diamond(run \rightarrow door\_open))}_{\Box A}; \underbrace{\Box(service \wedge \Diamond(run \rightarrow door\_open))}_{\Box A})}_{\Box A; \Box A}$$
$$\rightarrow \underbrace{\Box(service \wedge \Diamond(run \rightarrow door\_open))}_{\Box A}$$

Listing 7.7: Application of the 2-to-1 property

Presenting consistency problems found in a specification by model checkers in a way that is fully comprehensible to a human process modeller is pretty difficult. Awad et. al [4] describe two reasons why: The first reason is that typically, a model checker gives counterexamples as a subset of the state space in the context of the automaton generated from the process model. Thus, a reverse mapping of these state sequences is not one-to-one on the modelling concepts used for the process model. The second has to do with the logical proof strategy of model checkers in general, which says that it is only necessary to find one counterexample to disprove a formula causing the running evaluation to stop if this one is found. However, many iterations would be necessary to free a specification with many business rules from all its contradictions. And a third reason based on the second one can be added. In some cases, it might be an inherent given that certain state spaces can practically never occur, even if they are theoretically allowed by the model. This is especially the case when the behaviour of real-world hardware (e.g., in a construction plant) is reflected by the model. When considering a very simple propositional non-error scenario of a fuel tank, it is not possible that it is both empty and full at the same time. If a process modeller wants to be formally correct, she would have to consider that very simple circumstance in her formal model, which might not be the desired way in terms of modelling efficiency as the models tend to become unnecessarily detailed and unmanageably large. However, efficiently overcoming trivial inconsistencies in large business rule specifications needs tool support [4].

7.5 Abstracting behavioural business rules

Evolving business rules over time often increase their degree of detail unnecessarily. At the final stage of this evolution, there are business rules that are too detailed and too complex to be comprehended. Thus, the motivation to scrutinize them decreases accordingly. To overcome this problem, a solution is introduced that is based on the idea of deducing the rules that are more general than the original ones, which is called *abstract rules*. In the best case, a business process specification turns out to be invariant. On the one hand, reducing process specifications makes it easier to manually check them for inconsistencies. On the other hand, it contributes to a better comprehensibility of the overall business process specification, on an abstract level. MONA is chosen for checking if a concrete original formula implies a certain abstract formula, which means that the following implication holds: $\vDash A_{concrete} \rightarrow A_{abstract}$. In our example, $A_{abstract} := \Box(p \rightarrow \Diamond q)$ can be taken from a set of formulas that have proven to be general in certain modelling cases, e.g., the liveness property used here. Such formulas can be regarded as a certain kind of universal common rules of a higher level of abstraction. The utilized method is the ISE subset of the ISEP (Introduction, Sequential Combining, Extending Leftward, Parallel combining) method, which in detail is described and proven by Moszkowski [77]. The left side of the implication ($A_{concrete}$) is the actual detailed business rule to be simplified (see subformula F1.0 of 7.1). ISE can be applied to the following basic rules: First, the abstract rule to apply also has to obey the 2-to-1 property, which means that the following must hold: $\vDash (f_{abstract}; f_{abstract}) \rightarrow f_{abstract}$. Second, a formula of a subinterval must also be deducible to $f_{abstract}$.

The generic approach of applying ISE is described in Table 7.8. Please note that for the sake of completeness for presenting Moszkowski's method, parallel combining is included, although it is not utilized in the examples.

In the **Introduction** step, it has to be validated that for each concrete formula, the following holds: $\vDash A_{concrete} \rightarrow A_{abstract}$. The PITL formula represented by the $A_{concrete}$ specifies a behaviour that is unnecessarily complicated. Whereas, $A_{abstract}$ is the general

Introduction of a 2-to-1 formula	$\vDash A_{concrete} \rightarrow A_{abstract}$
Sequential combining of a 2-to-1 formula	$\vDash (A_{abstract}; A_{abstract}) \rightarrow A_{abstract}$
Extension of a 2-to-1 formula either left- or rightward	$\vDash (A_{concrete}; A_{abstract}) \rightarrow A_{abstract}$ $\vDash (A_{abstract}; A_{concrete}) \rightarrow A_{abstract}$
Parallel combining of arbitrary 2-to-1 formulas A	$\vDash (A \wedge A') \rightarrow A''$ (see example below)

Table 7.8: Generic description of the ISEP method

abstract and inferable (and also PITL) form of $A_{concrete}$. The example implication in Section 7.1 later shows that each subinterval formula can be seen as a concrete formula for which the introduction criterion holds. In a business rule context, the formula $\Box\neg p$ could mean that *it must never occur that* ($\Box\neg$) *the doors open while the car is moving* (p).

The **Sequential Combining** is the actual application of the 2-to-1 formula, remember $\vDash (A; A) \rightarrow A$. In the business rule context introduced above (Listings 7.1, 7.2, 7.3 and 7.4), this could mean that for any time a controller checks A, which is $\Box(p \rightarrow \Diamond q)$, the following must hold: *it is always the case that* $\Box$ (*if* ($\rightarrow^{1/2}$) *the doors open while the car is moving* (p) *then* ($\rightarrow^{2/2}$) *a warning lamp has to light in the driver's cockpit* (q) *afterwards* ($\Diamond$).

The **Extension** step of the ISEP technique is principally conducting the steps **Introduction** and **Sequential combining** to the adjacent subintervals.

Parallel combining is of the form $\vDash (A \wedge A') \rightarrow A''$. In a business rule context, this could, e.g., mean that

$$\underbrace{\Box(p \rightarrow \bigcirc p)}_{\text{A}} \wedge \underbrace{\Box(q \rightarrow \neg p)}_{\text{A'}} \rightarrow \underbrace{(\Diamond(p \wedge q) \wedge \bigcirc\neg(p \wedge q))}_{\text{A''}}$$

The real-world use case for this may, for example, describe a continuous signal p that is only interruptible by a second signal q that can occur only once at a time, and if q holds, p does not.

However, this step is not considered for the work presented here.

For simplifying a behavioural business rule based on Mozskowski's ISE(P) method, please consider the following example [77] containing the short symbols p and q denoting

business rule formulas to keep the overall formulas concise. Hint: In the following example the formula $\Diamond\Box q$ means that a state exists from which on q always holds. A shorthand is: q is sometimes stable.

$$\underbrace{((\Box\neg p);(finite \wedge fin\, p);(\Diamond\Box q);(finite \wedge fin\, q))}_{\text{F1.0}} \rightarrow \underbrace{(\Box(p \rightarrow \Diamond q))}_{\text{F1.1}} \tag{7.1}$$

The formula consists of four concatenated subintervals. The first states that within the subinterval, p is never met. The adjacent subinterval has finite length (indicated by the operator $finite$) and in its final state, p becomes true (indicated by the operator fin, read "fin"). This is followed by another interval that states that q toggles to a stable state for the rest of interval three. Moreover, our deduction is also valid when this interval is infinite as there is no length specified. The last subinterval finalizes the formula, stating that q is true (at least) in the very last state. The formal definition of the operator's **final state** ($fin\, A$) and **finite interval** ($finite$), according to Moszkowski's definition [77] are: $fin A \equiv \Box(empty \rightarrow A)$ and $finite \equiv \neg(true; false)$. The operator $empty$ is defined as $empty \equiv \neg \bigcirc true$, which simply means that the interval has length 0 and thus has only one state. The implication F1.0 $\rightarrow$ F1.1 may appear very obvious, however, the intention behind this example is to demonstrate how to systematically derive the right side of the implication from the left side by utilizing the 2-to-1 characteristics of the subformulas. The ISE acronym is constructed of the names of the following substeps. Please consider the following valid implications for the subformulas, the example is based on:

(1) $\vDash \Box\neg p \rightarrow \Box(p \rightarrow \Diamond q)$

(2) $\vDash \Diamond\Box q \rightarrow \Box(p \rightarrow \Diamond q)$

(3) $\vDash (finite \wedge fin\, q) \rightarrow \Box(p \rightarrow \Diamond q)$

The implications (1), (2) and (3) are evaluated as valid, with the help of MONA. Implication (1) is valid, in the case when p holds once the left side holds and thus the

entire formula holds. If p does never hold, the left side stays false and the right side always holds. The state of q has no consequence of the validity of the formula. The implication (2) holds, should the left side hold for q to be sometimes stable (if q holds, its state may never switch in the subsequent states). Once the left side holds in the case that q is sometimes stable, the implication of the right side holds in the case when p is true, anyway. Should p not hold in this case, the right side holds anyway in this state. Implication (3) is true if q is true in the last state (stated by the left side), which also holds for the right side since q shall hold at least once in the sequence of states. If p should not hold one on the right side formula, the right side is true anyway. If q does not hold in the last state, the left side gets true an thus the entire formula holds.

These implications are used for the reduction steps of the ISE-based example:

$$\underbrace{(\Box\neg p)}_{Sys_1}; (finite \wedge fin\, p); \underbrace{(\Diamond\Box q)}_{Sys_3}; \underbrace{(finite \wedge fin\, q))}_{Sys_4}$$

Introduction (see F1.0 of 7.1)

$$\rightarrow \Box(p \rightarrow \Diamond q); (finite \wedge fin\, p); \underbrace{\Box(p \rightarrow \Diamond q); \Box(p \rightarrow \Diamond q)}_{Sys_3 \text{ and } Sys_4}$$

Sequential combining

$$\rightarrow \Box(p \rightarrow \Diamond q); \underbrace{(finite \wedge fin\, p); \Box(p \rightarrow \Diamond q)}_{Sys_2 \text{ and } Sys_3; Sys_4}$$

Extending leftward, using (3)

$$\rightarrow \underbrace{\Box(p \rightarrow \Diamond q); \Box(p \rightarrow \Diamond q)}_{Sys_1 \text{ and } Sys_2; Sys_3; Sys_4}$$

Sequential combining

$$\rightarrow \Box(p \rightarrow \Diamond q)$$

The result of the ISE method is an abstract business rule that holds for the entire interval and says that q must occur at least once after p has occurred.

7.5.1 ITL representation in APRIL

The APRIL representation of the formulas used can be expressed as atomic formulas and definitions as shown below in Listing 7.10. Note that the definition of the formulas is kept general-purpose and not domain-specific, thus their syntax is close to the mathematical

ITL Formula	APRIL (alternative 1)	APRIL (alternative 2)
$\neg p$	not p	the negation of p
$\Box\neg p$	it must never occur that p	p is always false
$p \rightarrow q$	p implies that q	if p it follows that q
$finite \wedge fin\, p$	conclusion with p	p holds at the end
$\Diamond\Box q$	q is sometimes stable	a sometimes-stable q
$\Box(p \rightarrow \Diamond q)$	if p occurs q must occur	q triggered by p
$\Box(p \rightarrow \Diamond\Box\neg q)$	p has no effect on q	q does not react on p
$A; B$	A is followed by B	
$A_1; A_2; A_3; ...; A_n$	sequence of intervals where A_1 is followed by A_2 is followed by A_3 is followed by $\ldots A_n$	
$finite \wedge \Box[\neg]q \wedge \Diamond p$	while q $[holds\|isfalse]$ p occurs	even though p occurs q remains $[true\|false]$

Table 7.9: Custom atomic formulas for the car-door controller domain

reading and not very natural. As mentioned in the preceding sections, the framework can be extended and thus tailored to a specific domain using more specific statements that convey the exact intention by natural language. In this context, one of the two following alternatives can be seen as such an extension to the basic set of APRIL's atomic formulas. The APRIL framework is flexible enough to host several synonym patterns. Our experience with extending APRIL patterns is that from a natural language perspective, synonyms should be formulated as different parts of speech.

Definition
the negation of (p **as Boolean**) until (q **as Boolean**) is initializing
is defined as
sequence of intervals where p is always false followed by p holds at the end followed by a sometimes-stable q followed by q holds at the end.

Listing 7.10: APRIL representation of the introductory example of subformula F1.0 in Equation 7.1

p	0	0	**0**	**0**	x	**1**	**1**	x	$\mathbf{x_3}$	$\mathbf{x_3}$	x	x
q	x	x	$\mathbf{x_1}$	$\mathbf{x_1}$	x	$\mathbf{x_2}$	$\mathbf{x_2}$	x	**1**	**1**	x	1

Table 7.11: Example interval for definition in Listing 7.10

The signatures of the definitions in Listing 7.10 are mostly kept independent of a domain, which in this case, surfaces as an awkward-sounding statement with little natural language characteristics. But the intention here was to verbalize the pure ITL statement to demonstrate that even such mathematical statements can profit from being verbalized in APRIL. With the ISE method, it is possible to check whether the formula complies with a property of liveness. The Table 7.11 is an image of the model of the four intervals that pose the semantic body of the aforementioned definitions. Each interval has a length of three states, both F1.0 and F1.1 are true for (0 means $\neg p$, 1 means p, and x, x_1, x_2, and x_3 mean "doesn't matter"; adjacent-state variable values are printed in bold and are identical per definition). The reader may be aware of that the ISE method does not intend to find equivalent formulas, where F1.0 $\rightarrow$ F1.1 but F1.0 $\nleftarrow$ F1.1 (see Formula 7.1). Considering the truth table example below, it can be seen that F1.1 does not require the first "1" of interval three to hold, whereas F1.0 does. Please note that the pairwise states in bold denote adjacent states, each of which can be counted only once.

7.5.2 Indicators of the 2-to-1 property

According to Moszkowski, there is a set of prerequisite indicators to the 2-to-1 property, which is described and proven in [77]. Some are as follows:

1. Any state formula *w* without temporal operators in it (*w* is a generalization of the state variable *p*)

2. Any formula F with the only primitive temporal operator $\bigcirc$ (*next*) and no nested "*nexts*"; e.g., $\bigcirc \bigcirc F$ is not 2-to-1 (see also applications of Next-Logic [75]).

3. Any PITL formula of the form $\diamond C$

4. Any PITL formula of the form $C\ ;\ true$ (read initially C), which means that for some initial subinterval, C holds.

5. Any PITL formula of the form $w \rightarrow (B\ ;\ true)$ where *w* is a state formula and B is a PITL formula.

These indicators are met one by one in the following example scenarios:

1. Trivially, any invariant business rule, e.g., in predicate logic.

2. A is true at present, and A' is true in the next state.

3. Any state that must occur at least once in a sequence of states, e.g., *Box must pass the sensor gate* or *the chemical tank must stay empty* (here C is $\Box w$ and w states that the tank is empty).

4. Intervals that ought to be initially true, e.g., for describing regular pulses such as cycles, etc.

5. If-then rules

In Section 7.6, some of these indicators help to identify formulas complying with the 2-to-1 property for enabling ISE-based simplifications.

7.6 Evaluation of a practical use case

For the evaluation of the refactoring approach, a reference specification is chosen that has been developed by *Fraunhofer Institut für Experimentelles Softwareengineering* (eng. Institute of Experimental Software Engineering) in cooperation with the Daimler AG [46]. The specification describes the behaviour of automatic car door controllers according to Section 6.5 in the specification. [46]. The selection of the case study material was intended to evaluate business rules that are general enough to gain representative and comparable results. Hence, a more technical specification is chosen, as they typically emphasize sequencing and timing behaviour.

The primary aim of the case study was to find out if the ISE refactoring method presented in Section 7.5 is suitable to real-life business rule specifications. The second aim was to get an idea on the quantitative occurrence of complex business rules that can be potentially abstracted/simplified. In the example specification, the functional behaviour of several car systems is described. The systems considered for our case study are those for seat adjustment, door locks, and the window locks. We did not formalize the specifications for the user management, lighting of the interior, external mirror adjustment, diagnosis, error memory, and error codes. The overall number of pages for the functional specification section is 20, each of which contains an average of ten behavioural business rules. This amounts to a total of approximately 200 behavioural business rules. With our three investigated specification parts, we formalized round about 50 of them in Propositional Interval Temporal Logic to find out, how many of them comply with the 2-to-1 property and can eventually be simplified by the ISE method. For this purpose, only 6 business rules were sufficiently complex to be taken under consideration for a sensible demonstration of the ISE method. However, three of them were structurally different, which, strictly speaking, yields a ratio of 6% to be suitable for ISE and 12% that could be abstracted. Getting back to the aim of the case study, we can say that refactoring with ISE is suitable to real-life business rule specifications. Although the resulting abstracted rules are considerably smaller than the original business rules, it still is the responsibility of the requirements engineer to decide, whether the loss of detail is worthwhile in each individual case.

With respect to the specification abilities that the temporal extension of APRIL offers, it can be recorded that each of the business rules under consideration was specifiable in a way that is considered understandable to people with a basic understanding of (temporal) logic. About 70% of the business rules in the specification where comparable with complex business rules according to the definition of Section 8.2. Whereas, the remaining of 30% met the requirements of "simple" business rules, respectively.

7.6.1 Example refactoring for locking behaviour

The practical application of the method of the ISE-subset described in Section 7.5 is presented as follows in this section. An ISE simplification at the end of this subsection can be made for the formula. Therefore, consider the basic liveness property

$$\Box(close \rightarrow \Diamond locked),$$

which we want to introduce as an abstract formula for reducing the originally defined concrete formula

$$\underbrace{(\Box\neg close)}_{Sys_1}; \underbrace{(finite \wedge fin(close))}_{Sys_2}; \underbrace{(\Diamond\Box locked)}_{Sys_3}; \underbrace{(finite \wedge fin(locked))}_{Sys_4} \tag{7.2}$$

Here, $close$ denotes that a locking signal is detected ($CAN.ZV\_SCHL\_A = 10$) and $locked$ that the vehicle doors are locked ($VEHICLE\_LOCKED = 1$).

The initial behavioural business rule in 7.2 with the subintervals Sys_1 to Sys_4 states in the basic state Sys_1, the signal to lock the doors is not yet received. Sys_2 expects to detect the $close$ signal at least in the last state of the interval indicated by $fin(close)$. The adjacent interval states that the system must respond to by locking the doors (in Sys_3). Sys_4 may additionally be added in order to cover the mechanical characteristics of the system behaviour, e.g., when the mechanical lock bounces back for a short time, at least long enough to be detected by the controller. However, Sys_4 requires the lock state $locked$ to end up with $true$. The final implication yields the abstracted behavioural business rule. To support the ISE simplification, we also use the following valid implications:

$$\models \Box\neg close \rightarrow \Box(close \rightarrow \Diamond locked)$$
$$\models \Diamond\Box locked \rightarrow \Box(close \rightarrow \Diamond locked)$$
$$\models (finite \wedge fin(locked)) \rightarrow \Box(close \rightarrow \Diamond locked)$$

For any formula containing symbolic controller states, please consider their explanations in Table 7.13.

The ISE simplification for the formula 7.2 is:

$$\underbrace{(\Box\neg close)}_{Sys_1};\underbrace{(finite \wedge fin(close))}_{Sys_2};\underbrace{(\Diamond\Box locked)}_{Sys_3};\underbrace{(finite \wedge fin(locked))}_{Sys_4}$$

Introduction

$$\rightarrow \Box(close \rightarrow \Diamond locked);(finite \wedge fin(close));$$
$$\underbrace{\Box(close \rightarrow \Diamond locked);\Box(close \rightarrow \Diamond locked)}_{Sys_3;Sys_4}$$

Sequential combining (2-to-1)

$$\rightarrow \Box(close \rightarrow \Diamond locked);\underbrace{(finite \wedge fin(close));\Box(close \rightarrow \Diamond locked)}_{Sys_2;Sys_3;Sys_4}$$

Extending leftward

$$\rightarrow \underbrace{\Box(close \rightarrow \Diamond locked);\Box(close \rightarrow \Diamond locked)}_{Sys_1;Sys_2;Sys_3;Sys_4}$$

Sequential combining (2-to-1)

$$\rightarrow \Box(close \rightarrow \Diamond locked)$$

7.6.2 Example refactoring for safe unlocking

Another liveness property states that if the unlatching initially fails due to a battery voltage below 9 volts, an attempt to start the engine must be made. This can be formally represented by the APRIL behavioural rule in Listing 7.12.

The unrefactored APRIL rule with the name "*Try Start Engine At Low Battery*"

Behaviour
Try Start Engine At Low Battery **is defined as**
(LowBattery **and Sequence of intervals where**
LockedDoorState **is followed by**
DetectionOfTheOpeningSignal **is followed by**
FailedAttemptToOpenDoors) **implies that**
EngineStarts
with
LowBattery **is defined as**
always $(BatteryVoltage < 9)$,
LockedDoorState **is defined as**
the negation of p is sometimes stable ,
DetectionOfTheOpeningSignal **is defined as**
while q is $false$ p occurs,
FailedAttemptToOpenDoors **is defined as**
the negation of q is sometimes stable ,
EngineStarts **is defined as**
$KL\_START = 1$,
q **is defined as**
$S2.FT\_RIEGEL = 1$ and $S2.T\_RIEGEL = 1$,
p **is defined as**
$CAN.ZV\_SCHL\_A = 10$.

Listing 7.12: APRIL representation of the ITL example of formula 7.3

(see Listing 7.12) is defined as a conjunction of a proposition, describing that a state represented by the Symbol *LowBattery* always has to hold, with a sequence of combined subintervals materializing in the respective formulas represented by the symbols *LockedDoorState*, *DetectionOfTheOpeningSignal* and *FailedAttemptToOpenDoors*. The conjunction is combined with the proposition *EngineStarts* by an implication operator. This means the entire rule becomes valid if the formerly described conjunction holds, which implies that the formula described by *EngineStarts* must hold as well. Any of the used formulas (*LowBattery*, *LockedDoorState*, *DetectionOfTheOpeningSignal*,

CAN.ZV_SCHL_A	Input state received over controller area network (CAN) bus to lock/unlock all doors
VEHICLE_LOCKED	Internal state in the controller that reflects the locking state of all doors
KL_START	Output state used for undertaking a cold start of the engine.
S2.FT_RIEGEL	Input state that reflects the locking state of the rear doors
S2.T_RIEGEL	Input state that reflects the locking state of the front doors
BatteryVoltage	Input state that reflects the state of charge of the car battery from 0-15 Volts

Table 7.13: List of the state variables

FailedAttemptToOpenDoors, *EngineStarts*) is modelled as a local variable in the context of the APRIL rule indicated by the "*with*" keyword that separates the local variable definition section from the rule body. In the local variable declaration section, the *LowBattery* symbol for the respective local variable represents a temporal logic operator "*always*" that requires the proposition *BatteryVoltage* to be smaller than the integer constant 9 at any time. Whereas *BatteryVoltage* is a state variable symbol taken from the domain model. The domain model is simply described as a list of properties in the form of state variables in Table 7.13, which is trivially transformable into a single UML class. The following local variable *LockedDoorState* encloses a combination of atomic formulas described in Table 7.9, stating that at an arbitrary point in time, the embedded local variable p switches to (and remains) false. Decomposing local variables to atomic formulas and model symbols is straightforward for any of the local variables used.

For demonstrating the ISE method, it is reasonable to abbreviate the APRIL statements a bit using the basic domain concepts directly. Otherwise, the statements get too long and the recognition of the concepts introduced earlier is considered to be better for the reader. So, p is defined as $CAN.ZV\_SCHL\_A = 10$, which means that the signal to open the doors was received. Proposition q is defined as $S2.FT\_RIEGEL = 1 \wedge S2.T\_RIEGEL = 1$, which means that the rear and front doors are unlocked. Proposition b is defined as $BatteryVoltage < 9$, which means that the battery voltage is

less than 9 volts. This substitution yields the Equation 7.3 to work with for applying the ISE method.

$$(\Diamond\Box\neg p \wedge \Box b; finite \wedge \Box\neg q \wedge \Diamond p \wedge \Box b; \Diamond\Box\neg q \wedge \Box b) \rightarrow (KL\_START = 1) \quad (7.3)$$

The formula $\Diamond\Box\neg p \wedge \Box b$ represents the initial state, where the signal to open the doors was not received yet ($\Diamond\Box\neg p$) and the battery voltage is below 9 volts throughout each subinterval ($\Box b$). The adjacent finite interval $(finite \wedge \Box\neg q \wedge \Diamond p \wedge \Box b)$ represents the first step in the attempt to open the doors, stating that the doors are locked ($\Box\neg q$) and the signal to open the doors is received ($\Diamond p$). Remember, the assumption is that due to the low battery voltage, the doors cannot be opened. Thus, we use $\Diamond\Box\neg q$ to denote that any of the doors eventually remain in the locked state, even if the unlocking mechanism was able to temporarily pull the locks back to a position in which the door controller detects q for a single state in this interval. Finally, $KL\_START = 1$ means that a signal to (try to) start the engine is sent.

In the following reduction, it is shown how $(\Diamond\Box\neg p \wedge \Box b; finite \wedge \Box\neg q \wedge \Diamond p \wedge \Box b; \Diamond\Box\neg q \wedge \Box b)$ can be reduced to the liveness property $\Box(p \rightarrow \Diamond\Box\neg q)$ using its 2-to-1 characteristic. For this, please consider the following valid formulae:

$\models (\Diamond\Box\neg p \wedge \Box b) \rightarrow \Box(p \rightarrow \Diamond\Box\neg q)$
$\models (\Box\neg q \wedge \Box b) \rightarrow \Box(p \rightarrow \Diamond\Box\neg q)$
$\models (finite \wedge \Box\neg q \wedge \Diamond p \wedge \Box b) \rightarrow \Box(p \rightarrow \Diamond\Box\neg q)$

The previous refactoring yields the following formula:

$$\Box b \wedge \Box(p \rightarrow \Diamond\Box\neg q) \rightarrow (KL\_START = 1) \quad (7.4)$$

If we resubstitute p and q with the original statements

$$CAN.ZV\_SCHL\_A = 10$$

$$\rightarrow (\Diamond\Box\neg p \wedge \Box b; \underbrace{finite \wedge \Box\neg q \wedge \Diamond p \wedge \Box b}_{Sys_2}; \underbrace{\Diamond\Box\neg q \wedge \Box b}_{Sys_3})$$ **I**ntroduction

$$\rightarrow (\Diamond\Box\neg p \wedge \Box b; \underbrace{\Box(p \rightarrow \Diamond\Box\neg q); \Box(p \rightarrow \Diamond\Box\neg q)}_{Sys_2\ ;\ Sys_3})$$ **S**equential combining (2-to-1)

$$\rightarrow \underbrace{(\Diamond\Box\neg p \wedge \Box b; \Box(p \rightarrow \Diamond\Box\neg q))}_{Sys_1 \text{ and } Sys_2\ ;\ Sys_3}$$ **E**xtending leftward

$$\rightarrow \underbrace{(\Box(p \rightarrow \Diamond\Box\neg q); \Box(p \rightarrow \Diamond\Box\neg q))}_{Sys_1 \text{ and } Sys_2\ ;\ Sys_3}$$ **S**equential combining (2-to-1)

$$\rightarrow \Box(p \rightarrow \Diamond\Box\neg q)$$

(for detecting the unlocking signal) and

$$S2.FT\_RIEGEL = 1 \wedge S2.T\_RIEGEL = 1$$

(defining the lock states as *open*), it can be easily seen that the abstract formula now has the high-level business rule characteristic of that rule rather than the detailed original statement that is dedicated to an automation system. In natural language, the abstract rule states that: *it is always the case that if the signal to open the doors is detected and the locks do not switch to the open state, an attempt to start the engine has to be made.* Again, the APRIL representation of the unrefactored behaviour described in Formula 7.3 looks like in Listing 7.12, which is rather difficult to understand. Whereas, the refactored formula (see Equation 7.4) is represented by the APRIL rule in Listing 7.14.

We consider the refactored rule of Listing 7.14 more comprehensible than the original rule described in Listing 7.12. Please note that for the sake of simplicity, we have relocated the $\Box b$ in the subinterval formulas at the beginning of the APRIL statements for both rules (Listings 7.12 and 7.14), which is valid. The translation of the rule in Listing 7.14 into Tempura can be found in Listing 4.6.8. Please note that the transformation of the simplified PITL statement to its APRIL-representation has been conducted manually. However, the automated transformation of PITL to APRIL is possible since there is a one-to-one relation from the APRIL atomic formulas to their corresponding target language statements. This would additionally require a suitable heuristic for creating natural language compliant local variables and APRIL definitions, which is actually the opposite direction addressed in this work.

Behaviour
Try Start Engine At Low Battery **is defined as**
if LowBattery **and** DetectionOfTheOpeningSignal **has**
no effect on DoorLockState **it follows that**
EngineStarts
with
LowBattery **is defined as**
$(BatteryVoltage < 9)$,
DetectionOfTheOpeningSignal **is defined as**
$CAN.ZV\_SCHL\_A = 10$,
DoorLockState **is defined as**
$S2.FT\_RIEGEL = 1$ and $S2.T\_RIEGEL = 1$,
EngineStarts **is defined as**
$KL\_START = 1$,

Listing 7.14: APRIL representation of the ITL example of formula 7.4

7.7 Concluding on the abstraction of PITL-based business rules

The potential of using formal methods in practical requirements engineering is not exhausted. In the previous sections, it could be shown that some specifications describing behaviour tend to become very extensive. Although the implementation and their corresponding tests require rigour and a comparable degree of detail in the first place, some domain-related business rules can be addressed at a higher level of abstraction. This may reduce the detail to a still acceptable level while at the same time contributing to a disproportionately better readability and comprehensibility of the abstracted behavioural business rule. Presenting the abstraction of behavioural business rules was limited to those based on Propositional Interval Temporal Logic (PITL) since the axiomatic system behind the utilized ISE requires this. Another reason was to test if Moszkowski's novel methodology ISE is applicable to real-world specifications. The case study sought to show that such specifications contain enough PITL-based business rules that are 2-to-1 so that ISE is applicable anyway.

It is shown that certain chains of intervals can be reduced to simpler versions, e.g., formulas describing liveness. Although the utilization of ISE can reduce the complexity of large behavioural business rules tremendously, there are two major points limiting the applicability of ISE. The first is that the original business rule material has to allow the application of ISE. This is mostly determined if PITL-based rules comply with the 2-to-1 property or can be redefined to do so. Moreover, it also depends on the ability examine the business rule to deduce an abstract common rule from the original formulas in the subintervals. These abstractions have to be handcrafted as yet and depend on the creativity and the abilities of the requirements engineer to utilize model checking mechanisms and the corresponding tooling (e.g. PITL2MONA and MONA) in order to verify if the introduced simplifications are correct. Therefore, the presented list of indicators for detecting 2-to-1 can help in identifying the business rules to refactor. The second point is

that not every deduction makes sense in every context, and this the requirements engineer has to decide for each case individually. A different aspect that could be targeted for future work is evaluating whether the application of the methods presented in this chapter is suitable to meet the needs of the typical requirements engineer. Hence, the application of ISE intrinsically raises the level of complexity of the overall APRIL framework. It is yet to be discovered if the typical requirements engineer can manage the aforementioned challenges with a reasonable amount of training.

7.8 Summary

This chapter motivates using formal methods in requirements engineering to prove the correctness of APRIL statements describing rules of behaviour. By the checking of some logical properties (contradictions, partly contradictions, and redundancies), one can detect errors in APRIL statements and thus raise the quality of the set of formalized business rules. The second part of the chapter investigates a subset of a novel technique (ISEPI) invented by Moszkowski. The rationale of the ISEPI method is based on PITL, which is also introduced in this chapter. The ISE subset is applied to APRIL statements to raise the level of abstraction in behavioural business rules. As yet, its application is partly automated. Tool automation is implemented for using model checking in a form that PITL-statements in Tempura are transformed to a ws1s-compliant representation that can be interpreted by the MONA-Checker. The manual part requires finding suitable abstract 2-to-1 behavioural business rules that can substitute the original concrete business rules. Moreover, the back transformation of the simplified PITL version of a business rule into its corresponding APRIL-representation is also a manual task. Automating these aspects appears possible and can thus be issued to future work.

However, the aim of using the ISE subset is to make complicated, sequentially combined behavioural business rules more concise and better comprehensible by humans. A representative case study has been conducted to evaluate the practical harness to requirements engineers. The result based on the chosen domain was that the value potential of the simplifications has to be weighed out very carefully against the considerable effort made to apply the SEP method.

Chapter 8

Evaluation on the acceptance and applicability of APRIL

8.1 Introduction

This chapter presents the results of the investigation whether APRIL's core language can be used by persons with a basic understanding of logic and set theory and very little training in APRIL, as compared to its target language, OCL (see Section 8.2).

An additional case study based on the specification of the "Procedure Train Trip" of the European Union Agency for Railways [32] shows how APRIL is used to formalize a real-world specification written in a technical style. This is to investigate whether APRIL statements provide enough expressive power and additionally if pre-and post-conditions can be used for behavioural modelling (see the results in Section 8.3). Both case studies are devised to support the claim that the usage of an APRIL framework can narrow the gap between specification and implementation of software systems.

The fact that the core language of APRIL is based on OCL also helps to provide sufficient evidence that APRIL is expressive enough for real-world specifications. Regarding the work of, e.g., TU Dresden [115] and the Key Project [54], it seems valid that OCL is a formalism suitable for use in complex specification tasks occurring in industrial practice.

A comparison of APRIL with other textual and graphical languages, which are located in the context of model based testing, concludes the chapter.

8.2 Comparison of the expressiveness of the Syntax of APRIL and OCL

The goal of the case study was to discover whether the APRIL syntax can be understood by untrained users with a basic understanding of logic. For investigating this, a representative group of thirty computer science students in their first and second-year could be motivated to participate, of which the majority was completely inexperienced in the field of UML modelling and has never before heard of OCL. We considered OCL

version 2.0 for the benchmark language. That was because OCL 2.0 – as a part of the UML specification – is an established and well-defined language that closely suits to our purpose: defining business rules on UML-class models.

Two days before the case study, an information sheet was handed to the test persons, explaining the very basics of the APRIL and OCL syntax. This included predicate logic operators (e.g., universal and existential quantifier), operators on sets (e.g., for the union, exclusion, and intersection of sets), and the very important *join* operator. Moreover, the UML-class model concepts necessary to comprehend the APRIL and OCL materials were explained. This comprised the use of the most important of these, e.g., classes, associations, roles, and multiplicities. Directly before launching the case study session, a brief introduction into the domain of discourse was given on which the APRIL and OCL constraints were written against.

The case study sheet consisted of four sections. The first section dealt with questions on an example of a UML-class model and intended to demonstrate how mature the skills of the participants in UML modelling were, and further if the related UML-model essentials necessary to understand the tasks presented in the succeeding parts were comprehended. The second and third section required them to attempt to interpret and write down the meaning of a given sequence of 18 APRIL and 18 OCL constraints in their own words. These 36 constraints were based on an example UML-class model consisting of 5 classes. Each APRIL and OCL constraint had its semantic counterpart in the opposite language.

The complexity of the constraints increased rapidly, which can be examined in the Appendix E. In the last and shortest section, the test persons had to formulate OCL and APRIL constraints, based on business rules, given in real natural language (see also Appendix E).

The case study was carried out as follows. The group of test persons was divided into two equal-sized subgroups, each starting either with the third part (OCL) or the second part (APRIL). This was to counterbalance potential learning effects. Remember, each

constraint had its semantic counterpart in the opposite language and that is why learning effects could not be precluded.

The results are displayed in Figure 8.1, with respect to the average percentage of correct answers or correctly interpreted APRIL or OCL business rules.

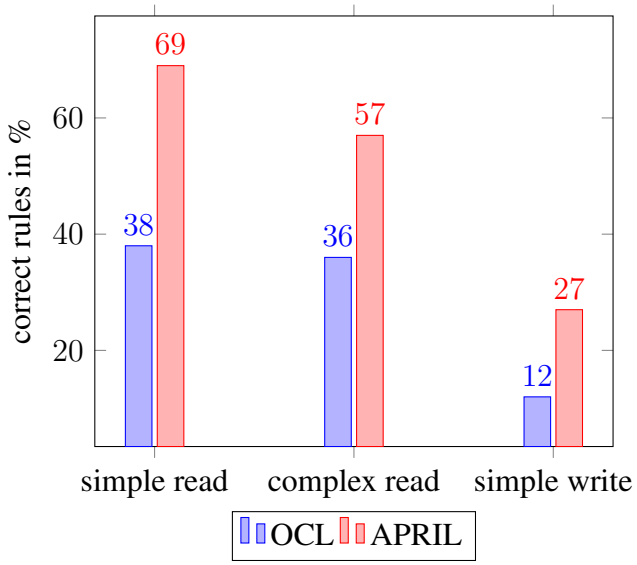


Fig. 8.1: Correctly interpreted/formulated APRIL/OCL business rules in %

Textual feedback of the test persons:

- About 50 per cent of the test persons spent less than 20 minutes for their preparation phase. About 30 per cent were unprepared. Persons of the remaining 20 per cent invested one to two hours to prepare for the survey.
- About 90 per cent of the test persons subjectively estimated that APRIL is more understandable than OCL. Whereas 2 students found both languages equally understandable and one student with a significant background in other formal languages found that OCL is more understandable.

The resulting percentage of APRIL, reflecting the correctly interpreted constraints, allows concluding that it is possible for untrained test persons to understand APRIL statements. A surprisingly high number of test persons was able to write rules. This discipline has been considered to pose a bigger problem regarding the low preparation

effort of 20 minutes for the major part of the participants. Students who invested more time for preparation gained better results in both interpreting and writing APRIL rules. For OCL we were not able to observe a similarly strong coherence between preparation time and improved results. The unexpectedly very good understandability of UML-class models (by 74% of correctly given answers with an average spread of 10%), even without any preparation, might be a good indication that the combination of a graphical notation to represent concept models and a textual notation for constraints is suitable for specifying understandable business rules.

The case study on the acceptance of APRIL only covers a comparison between APRIL and OCL. A comparison between APRIL and Tempura has not been conducted. We think that it is valid to assume that the language concepts of APRIL which allow making OCL statements more understandable also apply to those regarding Tempura. However, in the other direction, namely the formalization of behavioural business rules using the behavioural subset of APRIL, our hypothesis is that the required training effort is significantly higher for initially getting familiar with modelling behaviour and formalizing the respective rules. The reason for this is that the underlying ITL concepts that Tempura is based on are more complex than those of predicate logic that OCL is based on. Nevertheless, as outlined in the sections 2.5 and 2.6.1, Tempura is a suitable way of formalizing behavioural business rules at the industrial level.

8.3 Applicability study of APRIL based on ERTMS' "Procedure train trip" specification

This case study investigates the practical application of APRIL in a real-world scenario. Its aim is to show that, real-world specifications can be formalized using APRIL and remain understandable at the same time. Therefore, a representative subset of the specification of the ERTMS train model[1] is used, which is somewhat comparable to the synthetic business rules of Section 8.2 in terms of complexity. Hence, requirements engineers who are supposed to formalize the mentioned ERTMS subset must also have a basic understanding of logic and set theory. Given that the basic qualification of a requirements engineer is met and the ERTMS subset is formalizable in APRIL anyway and, at the same time, yielding statements comparably complex to those in the case study presented in Section 8.2, it may be valid to conclude that APRIL can be utilized for this kind of specification. Therefore, the expected result is that with a basic understanding of set theory and logic, it is possible to formalize the specification in an acceptable timeframe.

In order to provide evidence to the claim, a straightforward method for the case study is chosen. As a basis, the real-world "Train Trip" description of the ERTMS specification is chosen. The entire specification has been devised by the "European Union Agency for Railways" in order to set up a standard for computer-based train control. In order to obtain a simple metric for rating the results, a categorization for the complexity of the business rules is introduced, which is partly comparable to that of Section 8.2. The categories are "simple", "medium", and "complex", which is also described in detail in Table 8.1[2]. Based on the results of the acceptance study (Section 8.2), it can be recorded that the more complex APRIL business rules are, the more understandability can be provided to the entire specification. The specification has formalized as the methodology in Section

[1] see "Procedure Train Trip" specification part, described in Section 5 in [29] and Appendix F for the APRIL version

[2] An example-based mapping of the complexity criteria of the respective business rules is given in Appendix F.2

6.2 suggests. First, the used business concepts are formalized into a UML-class model, which in a second step, is then the basis for the business rule formalization into APRIL.

The special characteristic of the content of the "Train Trip" specification is that it is a verbalized version of a state automaton describing simple transition criteria and states after and before each transition. Therefore, the APRIL pre- and post-conditions were additionally chosen to investigate two things:

1. Comparing APRIL pre- and post-conditions with Tempura for specifying behavioural rules.

2. How to create the UML model without involving too many technical aspects of a state automaton, which would distract a reader from the actual business rule.

The aforementioned aspect under Point 2 outlines that if the state automaton style of the specification is transferred to APRIL, the transitions would be modelled as additional and synthetic class methods. Moreover, the states of the automaton would also require synthetic state variables. This may dilute the terse and concise specification style as additional APRIL statements would have to be introduced that describe technical aspects of the state automaton and not the actual domain knowledge. Preventing both while using pre- and post-conditions lets us concentrate on creating concise and understandable APRIL rules that are suitable enough for generating tests as the domain-related state variables are part of the specification. Hereby the first difference between Tempura and pre- and post-conditions surfaces. With Tempura, it is possible to describe different states in the context of the entire state automaton by using concatenated sequences. Whereas with pre- and post-conditions, the context is limited to the single transition determined by the class method. This means that with Tempura, it is possible to inherently model the control flow of the state automaton without introducing too much noise to the specification statements, which in the end, makes the tests more accurate.

However, regarding the results, it seems that pre-and post-conditions are suitable for this kind of specification if the intention is to focus on the domain knowledge. In terms of

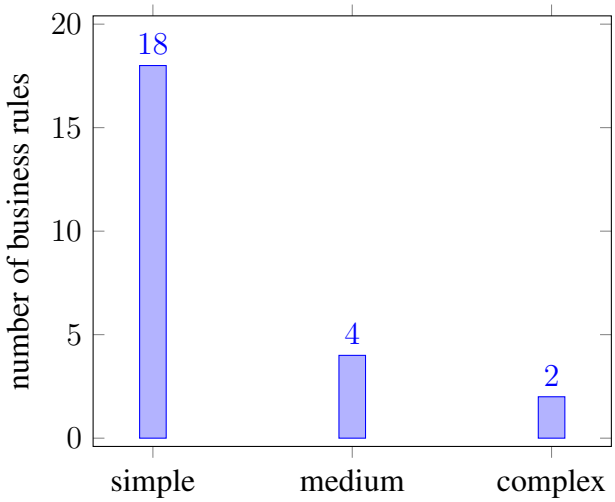


Fig. 8.2: Business rule complexity in the ERTMS specification.

expressiveness, it was possible to transform all 24 rules in the "Train Trip" specification into 22 APRIL rules and 2 APRIL definitions, which is as expected. The formalization can be examined in detail in Appendix F. It surfaces that the ratios of simple, medium, and complex business rules are distributed in favour of simple rules as Figure 8.2 shows. This is a bit unfortunate as APRIL's natural language expressiveness scales nicely with medium or complex rules. But this does not mean that APRIL is not suitable to be used for this specification type, especially because the acceptance study (see Section 8.2) shows that simple rules in APRIL are particularly well-understandable compared to OCL (see Figure 8.1).

Complexity criterion	Description
simple	The business rule contains a single operator or multiple concatenated operators, possibly nested to the depth of one
medium	The business rule uses one definition and/or contains multiple concatenated or nested operators
complex	The business rule uses more than one definition or local variable or contains multiple concatenated or multiple nested operators

Table 8.1: Description of the complexity criteria

The readability of the medium and complex business rules is good and somewhat

comparable to the readability of the natural language specification (see, e.g., Listing F.1.5 in Appendix F.1) if definitions are used in mixfix syntax.

Regarding the results, it is valid to conclude that, the APRIL language is suitable to be used in the kind of real-world specification presented.

8.3.1 Towards the application of the Methodology within the ERTMS context

This subsection gives an impression on the usage of the staged approach suggested by the methodology presented in Chapter 6. Therefore, the example specification of the ERTMS subset dealing with the "Procedure Train Trip" specification is used. As described, the methodology suggests the following steps to get from natural language specification to the APRIL statements.

1. Natural language refinement if necessary. Fortunately, the ERTMS specification does not need this linguistic beautification. Hence, this can be left to the relevant literature such as [98].

2. Extracting business concepts for the domain model

3. Abstracting and defining reusable sentence patterns

4. Formulating APRIL statements

In the following an example is constructed to get from the natural language business rule to the APRIL statement presented in 8.3.1. Therefore the Figures 8.3, 8.4, 8.5, 8.6 present copies of the original specification [29], which are taken as the basis for constructing (parts of) the domain model[3] in the first Step. Therefore, Table 8.2 presents the mapping between the business rule and the derived domain concepts. According to the methodology, this step is done manually. It surfaced that some business concept compositions have to be suitable for more than one related business rule. From a data

[3] see abbreviated domain model in Figure 8.7 and the full domain model in the Appendix in Figure F.1

modelling viewpoint for example, it can be more advantageous to model a property as an attribute rather than an association.

Creating APRIL statements from the original specification is sketched in the following subsections 8.3.1.1, 8.3.1.2 and 8.3.1.3.

8.3.1.1 Extracting business concepts for the domain model

In this section the transformation of the business rules into the domain model is sketched, serving as the basis for formulating APRIL rules from the figures 8.3 to 8.6.

From the noun-phrases in the business rules (figures 8.3 to 8.6), the domain concepts in Table 8.2 can be created, which finally materialize in the UML-classes in Figure 8.7. Each phrase relevant for the domain model and the business rule is written in *italics* on the left side of the table. For the sake of brevity, only those UML concepts are considered, which are relevant for the example business rule in Listing 8.3.1.

ID #	Requirements
S010	The ERTMS/ETCS on-board equipment is in one of the following modes: FS, LS, OS, SR, SB, SH, SN or UN When an event occurs, which leads to train trip reaction (**E015** – refer to chapter 4, transitions between modes), the process shall go to **A025**.

Fig. 8.3: Copy of the first part of the "Procedure Train Trip" subset of the ERTMS specification [29]

D130	If there is at least one pending emergency stop, the process shall go to **S130**. If there are no pending emergency stops the process shall go to **S140**.
S130	The ERTMS/ETCS on-board equipment waits for the RBC to revoke ALL pending emergency stops. When all emergency stops are revoked (**E135**) the process shall go to **S140**.

Fig. 8.4: Copy of the rules S130 and D130 of the "Procedure Train Trip" subset of the ERTMS specification [29]

ID #	Requirements
S140	The ERTMS/ETCS on-board equipment shall offer the possibility to the driver to select "start" (only if train data has been previously entered), or to select SH a) if the driver selects "start" and the level is 1 (**E150**), the process shall go to **S160** b) and the driver selects "start" and the level is 2 or 3 (**E155**), the process shall go to **S150** c) If the driver selects SH (**E145**), the process shall continue in the same ways as the procedure “Shunting initiated by the driver”. If the SH request is refused by the RBC (**E165**) the process shall return to **S140**.

Fig. 8.5: Copy of the rule S140 of the "Procedure Train Trip" subset of the ERTMS specification [29]

ID #	Requirements
D80	If the level is 1, 2 or 3 the process shall go to **A105**. If the level is 0 or NTC, the process shall go to **D085**
A105	The mode shall change to PT and the ERTMS/ETCS on-board equipment revokes the emergency brake command. For the supervision provided by the PT mode refer to SRS chapter 4. The process shall go to **D110**.
D085	If no valid Train Data is stored on-board, the process shall go to **A140** If valid Train Data is stored on-board, the process shall go to **D090**

Fig. 8.6: Copy of the rules D80, A105 and D085 of the "Procedure Train Trip" subset of the ERTMS specification [29]

Phrase in Business Rule	Derived Domain Concept(s) in class model 8.7
"The *ERTMS/ETCS on-board equipment* is in one of the following *modes: FS, LS, OS, SR, SB, SH, SN or UN*" (S010)	• Class with name *EEOnBoardEquipment* • Attribute *mode* of type *TrainMode* in class *EEOnBoardEquipment* • Class with name *TrainMode* • Attributes *FS, LS, OS, SR, SB, SH, SN and UN* of type *Boolean* in class *TrainMode*
"The *ERTMS/ETCS on-board equipment* shall offer the possibility to the driver to select *"start"* (only if train data has been previously entered), *or to select SH*"	• Method *Start* in class *EEOnBoardEquipment* • Method *SetModeToSH* in class *EEOnBoardEquipment*
"If no valid *Train Data* is *stored on-board*, the process shall go to A140"	• Class with name *TrainData* • Association from class *EEOnBoardEquipment* to class *TrainData* with rolename *trainData* and cardinality 0-to-n on the *TrainData* side.
"The mode shall change to PT and the ERTMS/ETCS on-board equipment *revokes* the *emergency brake* command."	• Attribute *emergencyBrakeActive* of type Boolean in class *TrainData*
"If the level is 1, 2 or 3 the process shall go to A105."	• attribute *level* of type byte in class *EEOnBoardEquipment*
"The ERTMS/ETCS on-board equipment shall offer the possibility to the driver to select "start" (only if train data *has been previously entered*), or to select SH"	• Attribute *previouslyEntered* of type Boolean in class *TrainData*

Table 8.2: Derived business rule concepts

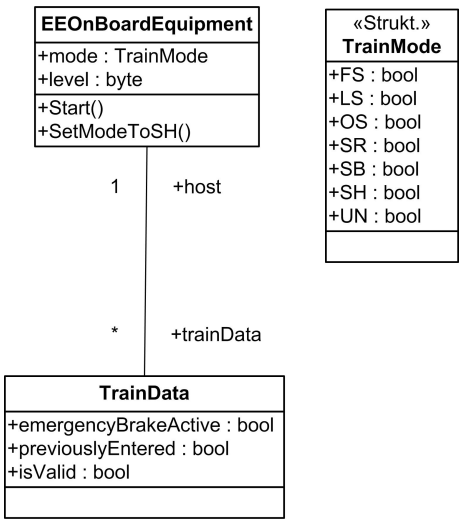


Fig. 8.7: Abbreviated Domain Model of the ERTMS / ECTS Train System for the Driving Use Case (see Figure F.1 showing the complete model)

8.3.1.2 Abstracting and defining reusable sentence patterns

Reusable statements in APRIL can be mapped either to local variables or functions or if the statement shall be reused in multiple business rules it is reasonable to choose a mapping to a definition. With the business rule in Figure 8.4 we have exact this situation. The rule makes assumptions on the system's behaviour based on the existence of "pending emergency stops". This is modelled as an entity of train data within the board equipment, indicated by the association of the class *EEOnBoardEquipment* and *TrainData* (in the model shown in Figure 8.7). Choosing a suitable representation for the intended meaning of the business rule by means of APRIL definitions can be a creative task. This is because the readability of the inline usage of the definition embedded in the business rule statement shall be as close to natural language as possible. Hence, the signature of the APRIL definition has an outstanding role because this is the means to transport the semantics of both worlds, natural language and formal logic. This is where mixfix naming brings its full strength to bare and the reason why mixfix is part of the design of the APRIL language.

The chosen sentence pattern formalized as signature of the APRIL definition (see

Listing 8.3.1) expresses that a pending emergency stop in the board equipment exists:

pending emergency stops in (equipment as EEOnBoardEquipment) exist

The APRIL definition takes one argument named *equipment*, which is of type *EEOnBoardEquipment*, reflecting the relation to the respective class in the domain model. In the body of the definition the actual business rule is formulated as existential quantification, that checks if there exists a *TrainData* entity in the related set referenced by *equipment.trainData* holding *true* in its property *emergencyBrakeActive*. The entire definition itself is specified as a predicate, indicated by a return type *Boolean*, stating that the business rule in the body yields either *true* or *false*. In this case *true* means that there are pending emergency stops in the board equipment and *false*, there are none.

8.3.1.3 Formulating APRIL statements

This subsection demonstrates how reusable statements can be applied in a business rule context and explains the rationale behind the example in Listing 8.3.1.

From the specification S130 and S140 a) in Figure 8.4 it follows that the method *start* of the board equipment can only be called if certain preconditions are met. These preconditions are

1. All emergency stops are revoked.
2. Attribute level of the board equipment is set to value 1
3. train data has been entered previously

These preconditions can be formalized in the context of the domain model as follows:

1. using the definition of Listing 8.3.1 with an instance of the board equipment object symbolized by the parameter *this EEOnBoardEquipment*, which looks like: "*pending emergency stops in* ***this EEOnBoardEquipment*** *exist*".
2. *level = 1*

3. *trainData.previouslyEntered = true*

For formalizing the specification, a precondition named *S130_140* of the method *start* of the class *EEOnBoardEquipment* has been devised (see Listing 8.3.1). In the body of this precondition the afore mentioned sub statements are concatenated by the *and*-operator since the specification requires all three propositions to hold.

For the precondition S140 in Listing 8.3.1 the corresponding specification requires that the *mode* of the board equipment can only be set to *SH* if there are no pending emergency stops. The formalization is simply done by defining a precondition for the method *SetModeToSH* of the class *EEOnBoardEquipment* and reusing the APRIL definition ("*pending emergency stops ... exist*") within a negation operation, indicated by the APRIL statement "*it is not the case that* ***pending emergency stops ... exist***".

```
1 Definition pending emergency stops in (equipment as
    EEOnBoardEquipment) exist yielding boolean is defined as
2 at least one data in equipment.trainData satisfies that
    emergencyBrakeActive = true.
3
4 Precondition S140 concerns EEOnBoardEquipment.SetModeToSH() is
    defined as
5 it is not the case that pending emergency stops in this
    EEOnBoardEquipment exist.
6
7 Precondition S130_140 concerns EEOnBoardEquipment.Start() is
    defined as
8 level = 1 and trainData.previouslyEntered = true and it is not
    the case that pending emergency stops in this
    EEOnBoardEquipment exist.
```

Listing 8.3.1: ERTMS / ECTS Train System, Use Case "Train Trip" Rule 5

8.3.2 Example on translating APRIL to OCL, a sketch

Once the APRIL statements are defined, the APRIL framework prescribes the following steps to transform these statements into the respective target language. In the case of the concrete example rule *S130_S140* from Listing 8.3.1, an APRIL precondition in combination with a definition is used, which determines the target language to be OCL.

According to APRIL's grammar, (in Section 4.6.1) the abstract syntax tree of the APRIL precondition *S130_S140* is constructed as sketched in the upper part of Figure 8.8, which is done before the mapping of the AST-nodes to their corresponding templates is started. Please note that for the sake of brevity, those production rules are left out (indicated by the dotted node "...") that do not cater to the comprehensibility of the AST-building mechanism. Furthermore, the example abstains from resolving the inner definition statement since this would not add new aspects for understanding the transformation mechanism. The lower part of Figure 8.8 shows how each node from the abstract syntax tree is mapped to its corresponding instantiated string template. A detailed description on how the substitution mechanism works in general, is given in Section 5.3. The final step is to integrate the values of the fully parameterized templates. Therefore, a subordinate template provides the OCL-text to be inserted into the parameter placeholder for its superordinated template. After integrating the root template, the translation is completed. The finished OCL code for the example business rule can be examined in Listing 8.3.2.

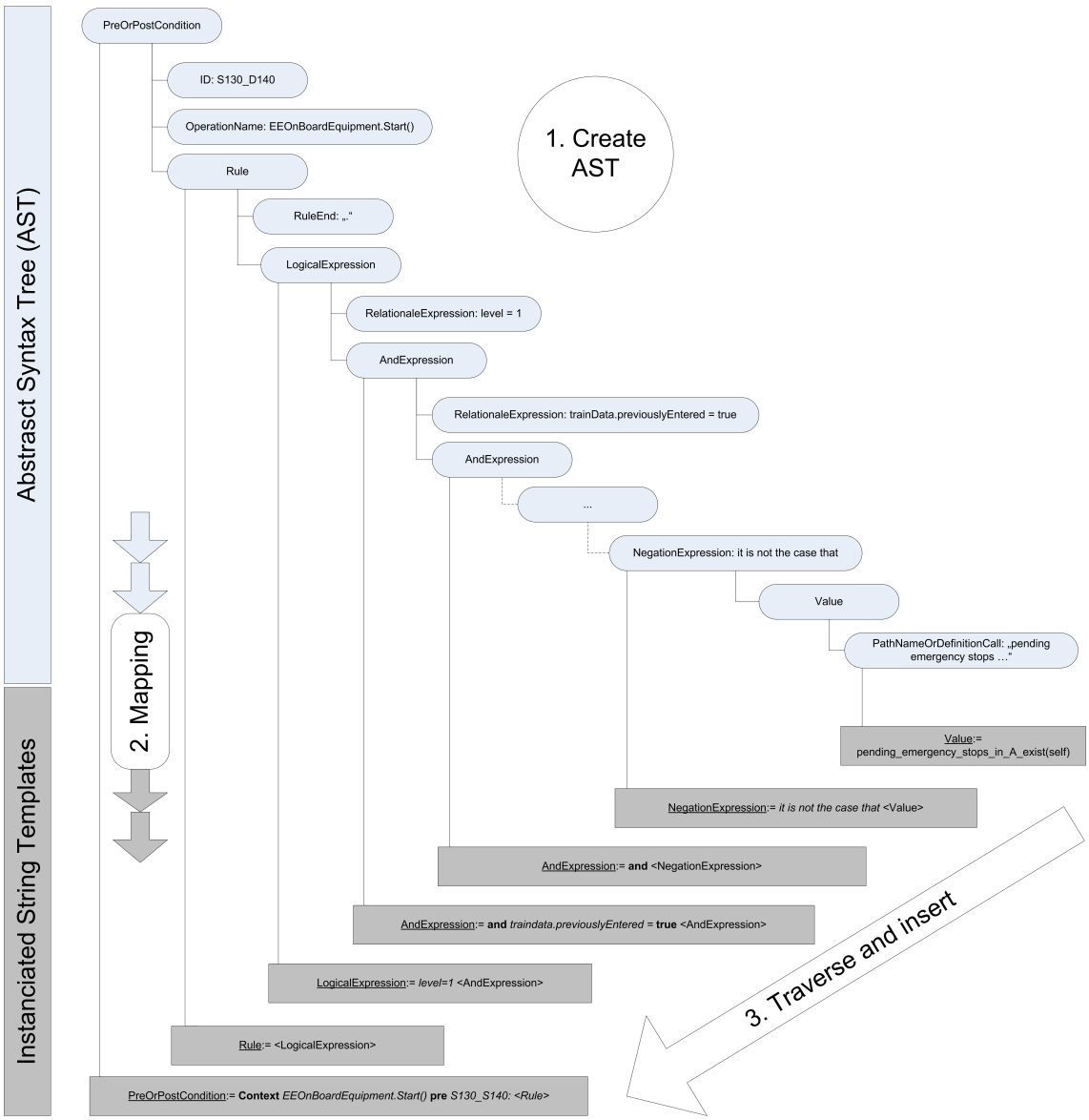


Fig. 8.8: Sketch of the translation steps for the APRIL precondition "S130_S140" of Listing 8.3.1

```
context EEOnBoardEquipment
def: pending_emergency_stops_in_A_exist(a : EEOnBoardEquipment)
   : Boolean =
a.trainData->exists(data | data.emergencyBrakeActive = true)

context EEOnBoardEquipment::Start() pre S130_140:
self.level = 1 and self.trainData.previousEntered = true and not
   (pending_emergency_stops_in_A_exist(self))

context EEOnBoardEquipment::SetModeToSH() pre S140:
not (pending_emergency_stops_in_A_exist(self))
```

Listing 8.3.2: OCL representation of rule 8.3.1

8.4 Application of the Methodology and Test Code Execution

This section gives a brief impression on how the usage of APRIL surfaces to the user by outlining the introductory steps for formalizing business rules in APRIL. Achieving a continuous work flow using the APRIL framework, it is necessary to describe the pre-preparation and evaluation steps around the automated part shown earlier. Therefore, the subsections 8.4.1 and 8.4.2 try to make clear the problem using concrete examples. This shall round the abstract methodology off, that gives a general guideline on (1) the initial manual specification refinement and (2) the result evaluation. Both, the initial manual specification as well as the result evaluation are set into the context within the methodology, which can be examined in Figure 6.1 of Chapter 6. Please note that for a better understanding of the materials especially for natural language refinement, a domain is addressed that is more business like.

8.4.1 Natural Language Refinement

Before starting the definition of business rules in APRIL, the specification has to be in a certain mature state that might need some manual natural language pre-processing. A second important aspect is to be able to derive the UML-class model (see Figure 8.9) using the typical object oriented requirements analysis methods.

An unrefined business rule may state that

```
1 High potentials are paid better.
```

Problem is that the nominalization "high-potentials" is not defined specifically and may lead to invalid implicit assumptions on its meaning. In order to resolve the nominalization, the term gets refined in the context of the domain UML-class model, which says that "high-potentials" are employees (*class Employee*) that work on more projects compared

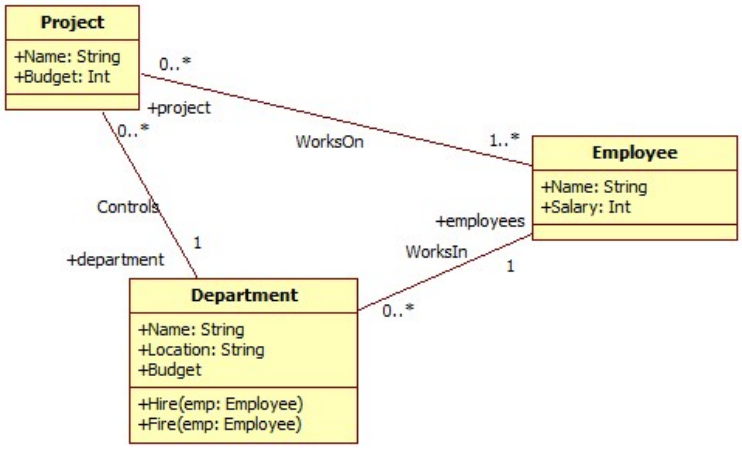


Fig. 8.9: UML Model as Result of the Requirements Analysis Step

to others. A second issue is introduced by the verb phrase "get paid". In the context of the domain model this should be rewritten to "get a higher salary", since the class *Employee* has the attribute *salary* that is of type *Int*, which can be used as an operand in a relational operation (e.g. ">="). A third issue is that in the verb phrase its object is missing. The intention of the business rule was to state that those employees that work on more projects shall get a higher salary than those who work on fewer projects. Therefore, the business rule shall be rewritten accordingly.

The natural language refinement rules used here are described in their general nature in Section 6.3.1 of Chapter 6. However, the refinement rules applied to the example here can be found in the Table 6.1 and are applied as follows:

- Rule (row 1): resolve imprecise nominalization (*"High potentials"*) by using the model related term *"Employees that work on more projects"*
- Rule (row 2): rewrite imprecise verb phrase (*"paid"*) by using *"get a higher salary"*.
- Rule (row 3): add missing object (relative pronoun) into the verb phrase (who is paid better than who?). Term *"than those who don't"* is introduced.

Refined natural language

```
Given two different employees it shall be that the employee who
   works on more projects than the other shall be paid a higher
   salary.
```

The APRIL transformation of the business rule is:

```
Invariant MoreProjectsHigherSalary concerning Employee:
every employee1, employee2 in all instances of Employee
   satisfies that higher salary is paid for employee1 if he
   works on more projects than employee2 .

Definition higher salary is paid for (employee1 as Employee) if
   he works on more projects than (employee2 as Employee)
   yielding boolean is defined as
number of employee1.projects > number of employee2.projects
   implies that employee1.salary > employee2.salary .
```

Other Business rule based on Figure 8.9 may be as follows:

- Rule **EmployeesInControllingDepartment**

  Unrefined natural language

  ```
  Cross department elaboration shall be limited.
  ```

  Applied natural language refinement rules from Table 6.1:

  - Rule (row 1): resolve nominalization (*Cross department elaboration*)

  Refined natural language

```
Employees working on a project must also work in the
   controlling department
```

APRIL

```
Invariant EmployeesInControllingDepartment concerning
   Project is defined as
Those employees working on a project must also work in that
   department .

Definition Those (employees as Set(Employees)) working on a
   project must also work in that (departement as Department
   ) yielding boolean is defined as
employees is in departement.employees .
```

- Rule **BudgetWithinDepartmentBudget**

  Unrefined Natural language

```
A project cannot spend more than the controlling department
   can provide.
```

  Applied natural language refinement rules from Table 6.1:

  - Rule (row 2): Refinement of the meaning of the verb, subject and object of the verb phrase.

Refined natural language

```
The budget of a project must not exceed the budget of the
   controlling department.
```

APRIL

```
Invariant BudgetWithinDepartmentBudget concerning Project is
    defined as
The budget must not exceed the department.budget .

Definition The (number1 as Int) must not exceed the (number2
    as Int) yielding Boolean is defined as
number1 <= number2 .
```

- Rule **MoreEmployeesThanProjects**

  Unrefined natural language

  ```
  Each project has to be equipped with appropriate manpower.
  ```

  Applied natural language refinement rules from Table 6.1:

  - Rule (row 2): imprecise verb phrase *equipped*.
  - Rule (row 3): imprecise substantive *manpower*.

  Refined natural language

  ```
  The number of projects of a department must not exceed the
     number of employees in that department.
  ```

APRIL

```
1 Invariant MoreEmployeesThanProjects concerning Department is
      defined as
2 The number of projects must not exceed the number of
     employees .
```

Please note that the APRIL definition *The (number1 as Int) must not exceed the (number2 as Int)* defined in the listing before is reused here.

8.4.2 Test Code Execution

In this section a brief idea is given how the generated APRIL statements are applied to the historical data output of the productive system. Therefore, this section shows how the tooling for the evaluation of business rules can be integrated in the validation process. For executing behavioural business rules AnaTempura is used, since it is the only tool that is able to execute Tempura statements and the interested user may consult the technical documentation[4]. Whereas, with OCL there are several alternatives (e.g. Bremen USE-Tool, Dresden OCL-Toolkit, Borland together, etc.). For a good understanding of the material and an appropriate handling, the Bremen USE-Tool was chosen, since it was devised also for educational reasons and allows object animation, which is helpful to reconstruct how the checking mechanism works.

Unlike the approaches around model based testing , in which test cases are generated automatically, APRIL does not prescribe the way for gathering the test cases. However, the method to gather the test data here is to execute the target system in a defined test case and record the test data accordingly, that is interpretable for the execution system that is determined to do the evaluation. In the example case, the Bremen USE-Tool is chosen and Section 8.4.2.2 gives an impression of how the test data has to be formatted. The data has

[4]there is also a brief introduction on AnaTempura by Anotnio Cau, see http://www.antonio-cau.co.uk/ITL/publications/reports/anatempura.pdf

to comply with the domain model, which for USE is exemplified in Section 8.4.2.1. For concluding on the test results it is necessary that the test case (if conducted automatically or manually) can be associated with the test data and the involved Business Rules, which take the role of the test. As mentioned, APRIL does not prescribe a method to keep the triple of test case, test code and historical data together. In the opinion of the author, this should be imposed by the development process and not vice versa, in order to keep the application of APRIL-framework as flexible as possible.

8.4.2.1 The Model for the Target System

The USE-Tool can execute OCL-constraints based on a UML-Model. As an example representation of a UML-model, the listing below holds some textual UML-model definition that comply with the graphical UML-model shown in Figure 8.10, which is also a screenshot of the USE-Tool. The definition of classes and attributes is straight forward and does not need to be explained any further. Associations between classes are modelled as separate entities introduced by the key word "*association*" followed by the name of the association. The body of an association description holds a list of the association ends denoted by their names and a cardinality declarator. For example *Employee[*]* means that the association end at the class Employee has the cardinality 0-to-many. OCL-constraints are imported and processed without any rewriting. They just have to comply with the syntax defined in the UML standard for OCL. The example constraint states a business rule that requires that the number of employees of a department (*self.employee->size*) shall be greater or equal (>=) to the number of projects driven by that department (*self.project->size*).

The displayed examples (see Listing 8.4.3) are taken from the online documentation of the USE-Tool[5].

[5]http://www.db.informatik.uni-bremen.de/projects/USE/prepost.html

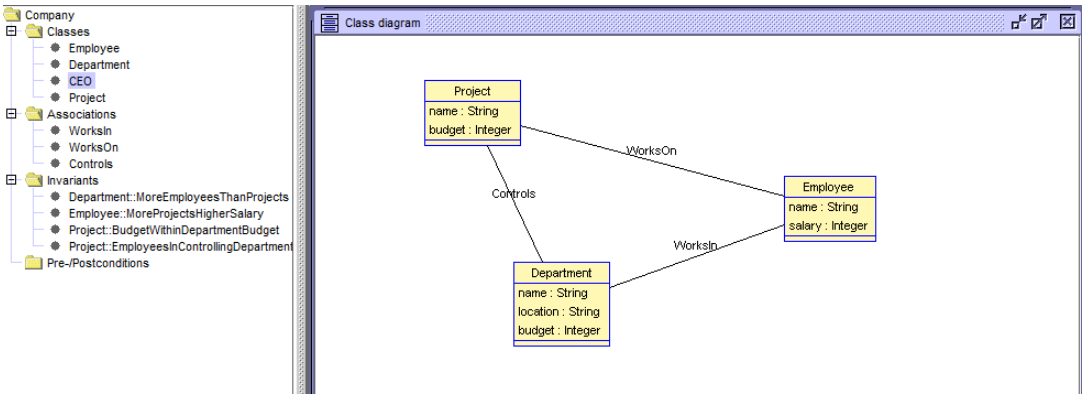


Fig. 8.10: Image of the Use Tool with the Imported UML-Model

```
class Employee
attributes
  name : String
  salary : Integer
end
class Department
attributes
  name : String
  location : String
  budget : Integer
operations
  hire(p : Person)
  fire(p : Person)
end
...
association WorksIn between
  Employee[*]
  Department[1..*]
end
...
context Department
  inv MoreEmployeesThanProjects:
    self.employee->size >= self.project->size

context Employee
    -- employees get a higher salary when they work on
    -- more projects
  inv MoreProjectsHigherSalary:
    Employee.allInstances->forAll(e1, e2 |
```

```
30        e1.project->size > e2.project->size
31          implies e1.salary > e2.salary)
32
33  context Project
34      -- the budget of a project must not exceed the
35      -- budget of the controlling department
36    inv BudgetWithinDepartmentBudget:
37      self.budget <= self.department.budget
38
39      -- employees working on a project must also work in the
40      -- controlling department
41    inv EmployeesInControllingDepartment:
42      self.department.employee->includesAll(self.employee)
43
44  context Department::hire(emp : Employee)
45    pre  hirePre: employees->excludes(emp)
46    post hirePost: employees->includes(emp)
47
48  context Department::fire(emp : Employee)
49    pre  firePre:  employees->includes(emp)
50    post firePost: employees->excludes(emp)
```

Listing 8.4.3: UML / OCL representation in the USE-Tool

8.4.2.2 Applying the Business Rules on the Historical Data

Once the productive code stepped through its logic that ought to be addressed by a test case, the gathered historical data can be presented in the form described by the listing below to the USE-Tool. This is actually a textual representation of an object model based on the classes specified in Figure 8.10. The idea behind how the USE-Tool evaluates

OCL-constraints is to reconstruct parts of the system state, that are related with the domain concepts. Therefore, the object population is created using the USE-Tool commands shown in the listing below. Here the *!create*-command is used to instantiate an object with a name and a type, both separated by a colon (e.g. *department1:Department*). The *!insert*-command is used to instantiate an association for linking objects. E.g. *!insert (emp1,department1) into WorksIn* links an object of tpye *Employee* with the name *emp1* to a *Department*-instance named *department1*.

In the USE-Tool it is also possible to simulate the behaviour of class methods, by processing pre- / post-conditions. This is done by passing the following commands to the command input box of the USE-Tool steps:

1. Use the *!openter*-command (see listing below), passing the object name, on which the method shall be invoked with the corresponding (typed) parameters.

2. Emulate the side effects. e.g. in our example, it would be necessary to link the newly added emp5-object to an existing *Project*-object, using the *!insert*-command, respectively.

3. Invoke the *!opexit*-command to tell the USE-Tool to take the method from the call stack that has been pushed by the *!openter*-command in step 1.

Going through these steps, USE evaluates the pre-conditions before step 1 and the post-conditions after step 3.

```
1 !create department1:Department
2 ...
3 !create emp1:Employee
4 ...
5 !create proj1:Project
6 ...
7 !insert (emp1,department1) into WorksIn
8 ...
```

```
!insert (department1, proj1) into Controls
...
!insert (emp1,proj1) into WorksOn
...
!openter department1 hire(emp5)
...
!opexit
```

Listing 8.4.4: Object model creation with the USE-Tool

Once the historical data is imported as an object-model-representation (see Listing 8.4.4), the evaluation of the OCL-constraints can be conducted. Figure 8.11 show a result of an example model, which fails on the constraint named *BudgetWithinDepartmentBudget*. Failed constraints (given that the APRIL-business rules are specified as non-negative propositions) indicate that that part of the productive system does not comply with the specified business rule, which may be an error to be investigated, either on the business rule side or in the productive system.

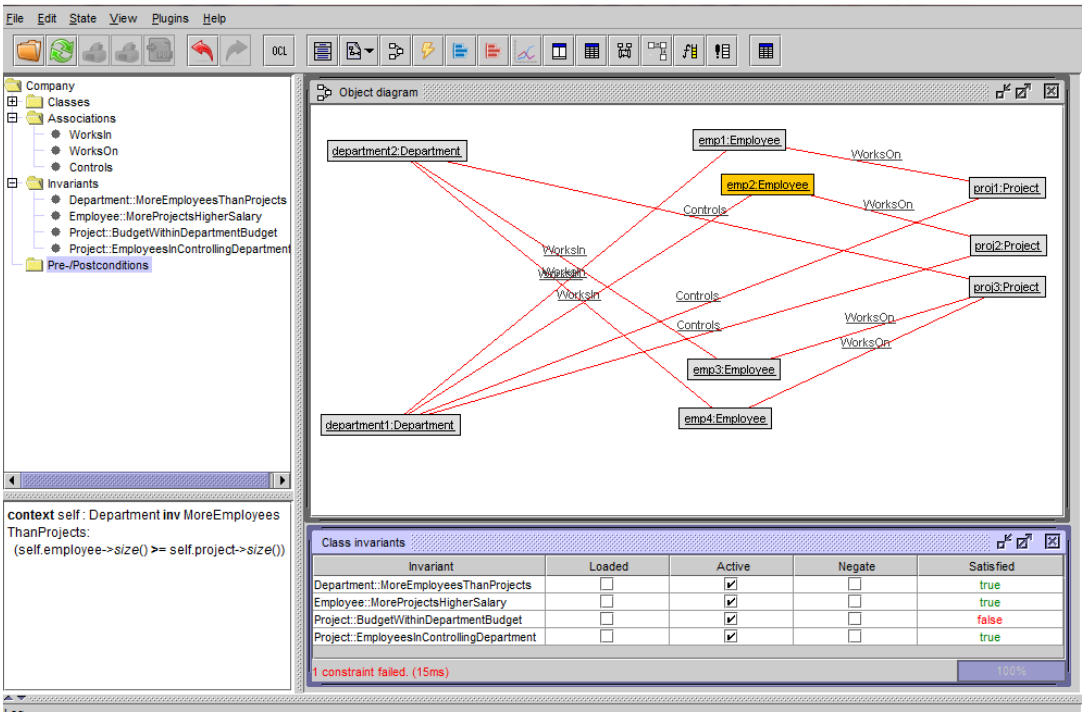


Fig. 8.11: Image of the Use Tool with an Animated Object Population

8.5 Comparison of Notations for Test Case Specification

This section contains a brief presentation of several frameworks (Alloy, SpecTRM-RL and UML-Activity Diagrams) that allow to specify business rules for automatically generating test cases. However, the particular interest is to highlight the differences of the notation forms to be able to conclude on their benefits and shortcomings compared to what is expressible with the APRIL language. Never the less, although APRIL does not address the automated generation of test cases, APRIL can be extended to also allow automated test case generation. The models that are utilized for the presentation are from two domains. Alloy uses a more business related domain, since on the one hand, especially Alloy shows its strengths in restricting sets, which is also an important aspect for business rules in this area but on the other hand comes pretty short in describing behaviour. For investigating the modelling capabilities of SpecTRM-RL the ERTMS train trip model is used[6], because of its particular strengths in describing the behaviour of controller components, which is the focus of the ERTMS model. Moreover, from the author's viewpoint SpecTRM-RL has serious flaws in expressing business rules as constraints on sets, which made it literally impossible to use it on the business related model. With the use of UML-Activity Diagrams as a third language, the aim was to present how the modelling of behaviour can be done with a widely used and standardized notation. For the investigation on UML-Activity Diagrams, the more business related model is used again.

8.5.1 Alloy

In Alloy the universe of discourse is modelled textually, which may be an advantage in small models compared to APRIL (or OCL), but with large models this tends to raise the complexity not only with respect to the model but also with respect to the business rules based on the model. Beahviour is modelled as operations on discrete global states.

[6] see Section 8.3.1 and appendix Chapter F

In terms of the modeling capabilities, some constructs consitiung the discourse universe of an ALLOY model are transofrmable to OCL (target language of APRIL). E.g. both APRIL and Alloy allow formulating arbitrary predicates.

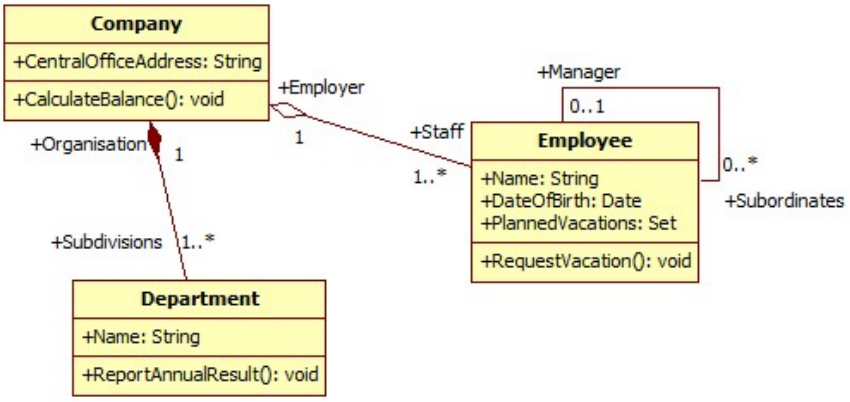


Fig. 8.12: Simple Business Domain Model (copy of 2.3)

Considering the simple model in Figure 8.12, which shows an UML model reflecting a discourse universe describing the concepts *Company*, *Department* and *Employee* an their associations, an ALLOY version would be modelled textually and look as follows (Listing 8.5.5):

```
1  one sig Company {
2    Staff : set Employee,
3    Subdivisions : some Department
4  }
5  sig Department {
6    Organisation : one Company,
7    Members : some Employee
8  }
9  one sig CEO extends Employee {}
10 sig Employee {
11   Employer : one Company,
12   Manager : lone Employee,
```

```
13    Subordinates : set Employee,
14    Salary : Int
15  }
16  one sig CEO extends Employee {}
17  // Any employee that is not the CEO is subordinate to the CEO
18  fact { Employee - CEO in CEO.^Subordinates }
19  // The CEO is managed by nobody
20  fact { CEO.Manager = none }
21  // A manager cannot manage himself
22  fact {all empl: Employee | empl.Manager != empl}
23  // An emplyee cannot be subordinate to himself
24  fact {all empl: Employee | empl not in empl.^Subordinates }
25  // any salary is bigger than 10000
26  fact {all s : Employee.Salary | s > 10000}
27  // returns a set of all subordinated Employees
28  fun allSubordinates[empl: Employee] : set Employee { empl.^
      Subordinates - empl }
29  // retruns true if the given Employee (empl) has more than two
      subordinates and earns more than 100000
30  pred LeaderGetsHighSalary[empl : Employee] { (#allSubordinates[
      empl] >= 2) => empl.Salary >=100000 }
31  // asserts that for any Employee that predicate
      LeaderGetsHighSalary holds
32  assert LeadingStaffEarnsALot {all empl : Employee |
      LeaderGetsHighSalary[empl]}
33  // asserts that none can manage himself
34  assert noneManagesHimself {all empl: Employee | empl in empl.^
      Subordinates }
35  // assert that the CEO is not subordinate to anyone else.
36  assert Ceo {all empl : Employee - CEO | not (empl.^Subordinates
```

```
    = CEO) }
// runs the assertions
check Ceo for 4
check LeadingStaffEarnsALot for 10 Employee
check noneManagesHimself for 10 Employee
```

Listing 8.5.5: Concept model in Alloy

Considering the afore presented listing, the alloy keyword "*sig*" denotes that a concept (comparable to a class in UML) shall be described. The concept name is followed by the keyword. In this example, the concepts *Company*, *Department* and *Employee* are introduced. Associations between the concepts are modelled as properties in a body block surrounded by curly brackets. Here, the concept *Employee* maintains three properties, which are *Employer*, *Manager* and *Subordinates*. Each property is typed, indicated by the colon character. Type information is provided by a cardinality and a type declarator that, in the case of modelling an association, is given by the name of a concept. For example the declaration of the property *Employer : one Company* in the body block of the *Employee* concept, says that it can hold a relation to **one** concept of type *Company*. Here, the keyword **one** is used to denote the cardinality of **exactly one**. Alloy allows to model additional cardinalities: **lone** denoting zero or **one**, and **some** for one or more.

Here are some other constructs in Alloy:

- Facts (denoted by the keyword **fact**) are constraints to the model that are comparable to invariants in APRIL and OCL. Facts are true on instances[7] of the model[8].

- Functions (denoted by the keyword **func**) are expressions that can be referenced by a name having a signature consisting of the name and parameters. A function can be invoked by providing expressions for its parameters. The type of an expression

[7] bindings of values to variables
[8] collections of instances

passed as a parameter has to comply with the declared type of the parameter declaration. The function definition requires a declaration of the return type.

- Predicates ((denoted by the keyword **pred**) are functions with the return type that can either map to *true* or *false*.

- Asserts (**assert**) model counter examples on instances. Hence, if an assertion has instances, a counter example to the defined facts is found an that assertion does not follow at least one fact.

- Checks (check) are used to execute assertions on the Alloy run time.

This chapter was in general devised to evaluate the practicability of APRIL. Under this spot, the following list of pros and cons outlines positive and negative aspects, allowing to the comparability to APRIL.

Pros:

- Very powerful set constraining using shorthand operators embedded in the Alloy's grammar (such as "^" for transitive closure or "*" for reflexive-transitive closure).

- Many constraints are expressible in predicate logic and a mathematical notation using set theoretic properties as well, which may be intriguing to mathematicians.

- Alloy runtime provides a graphical image of the counter examples found.

Cons:

- The universe of discourse is modelled textually, which is suitable for small models but becomes quickly incomprehensible with large models.

- The user has to be familiar with set theory. Significant parts of the expressiveness rely on directly using relation characteristics between sets (e.g. "^" operator for transitive closure or "*" for reflexive-transitive closure, etc.)

- The user has to be aware of the arity of a type, which raises the complexity of the expressions (e.g. the Alloy expression univ->univ denotes a cartesian product with a set of 2-tuples).

- Not suitable for describing behaviour.

8.5.2 SpecTRM-RL

SpecTRM-RL is a language to specify behavioural requirements with a state machine rationale. The language strives to cover any requirements specification aspect to support the complete, formal description of behaviour [59]. This is achieved by using a very elementary modeling basis (requirements state machine (RSM)) which as the authors [59] mention may be not suitable for complex specifications. RSMs are modelled as transitions with pre- and post-conditions as predicates, while the pre-conditions take the role of guard conditions for determining if the transition is to be entered or not. Post-condition predicates are evaluated after leaving the transition. SpecTRM-RL has been devised to overlay the quickly growing complexity of extensive RSM models, due to providing a better comprehensibility to the user.

To ensure the completeness of the specified systems, SpecTRM-RL uses a methodological meta model that is based on process control concepts. This involves a set of input variables, which get interpreted by a controller (modelled as a state machine), so that an appropriate set of correct output variables gets generated based on the met pre-conditions specified in the controller.

A very intriguing aspect of this methodological meta model is that the relationship between the output and the trigger that caused the triggering pre-condition to hold is maintained to allow the back tracking on certain outputs, which is useful for finding original causes e.g. in an error case.

Secondly, the occurrence of an input produces a tuple consisting the value of the input variable and an absolute time stamp. This makes it easily possible to go backward

and forward in a sequence of inputs, which is not possible with most temporal logic frameworks.

The inventors of SpecTRM-RL included several graphical notations to transport the semantics as understandable to a human reader as possible. As they also mention, there is no guaranty that SpecTRM-RL specifications are only reviewed by programmers or mathematicians with background in discrete mathematics, which lead them to the development of an easy to understand, set of graphical notation forms. Basically, the notation is adapted from the typical engineering drawing of a control loop with sensores at the input side, actuators at the output side, both measuring and manipulating the status of the process. Between the inputs and outputs, the controller evaluates the inputs and sets outputs accordingly to its internal logic.

Here are some examples based on the model shown in Figure 8.7:

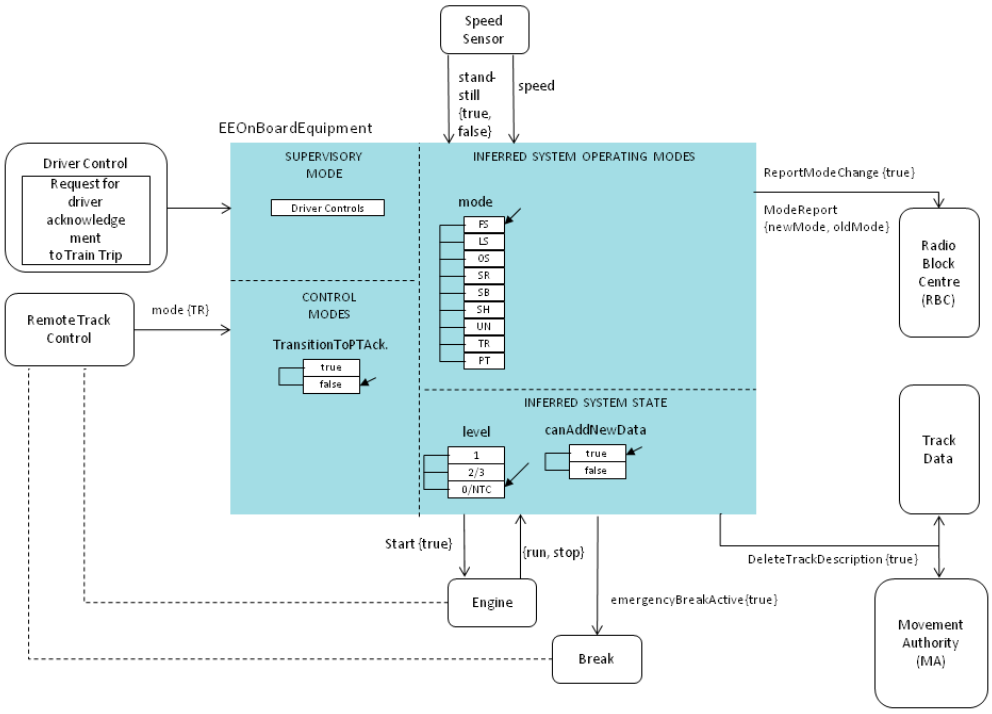


Fig. 8.13: ERTMS Example from S010 to S060 (see Figure F.2 in Appendix F.1)

The main components of a SpecTRM-RL specification are described in Figure 8.13, which is taken from the train trip case study introduced in Section 8.3. On the top left part

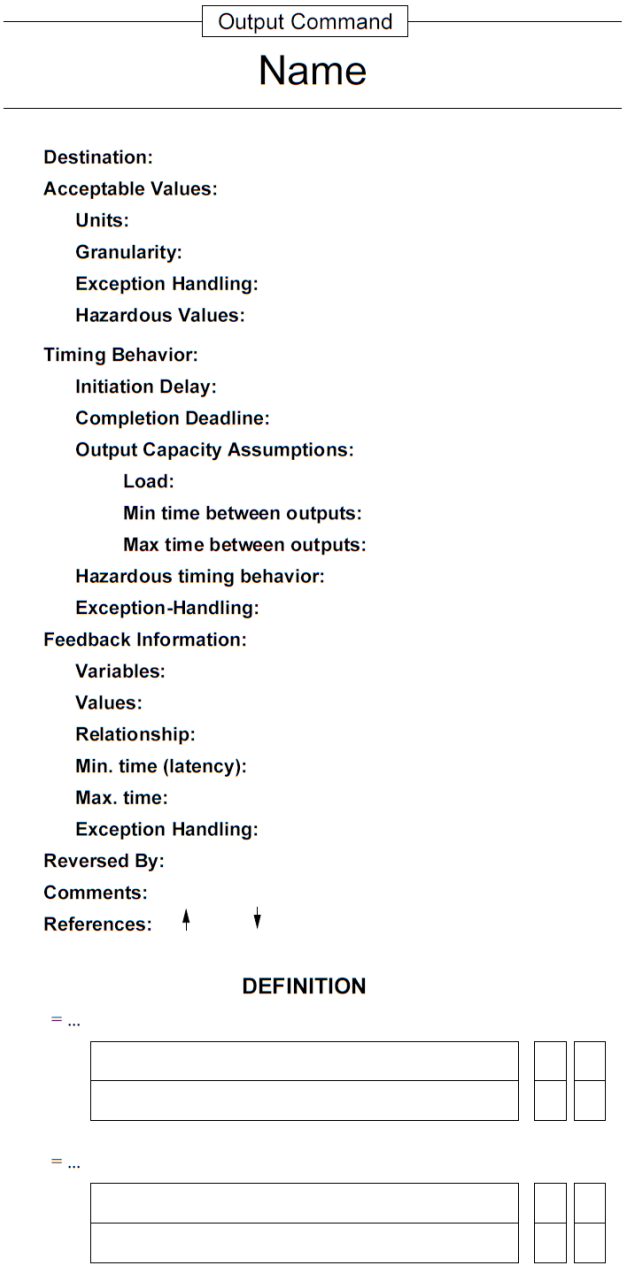


Fig. 8.14: Template for a Definition Sheet of an Output Command in SpecTRM-RT (see [59])

of the blue rectangle, the supervisory mode of the controller of the EEOnBoardEquipment (EEBE) is modelled, in this case it only contains the *Driver Controls*. The bottom left quadrant contains the operating modes of the controller, which is for presenting the relevant internal states of the controller to an external entity (e.g. a Human Machine Interface (HMI)). The top right corner models the *Inferred System Operating Modes*, the

Fig. 8.15: Definiton of a transition condition on SpecTRM-RT

EEBE can be in. The *Inferred System Operating Modes* describes the super ordinate state model for the *Inferred System State*. In complex system descriptions the super ordinate state model can itself be a controller of another process. The task of the *Inferred System State* is to describe the system states of the controller, here the EEBE. Any default state is indicated by an arrow pointing at the default value. In this context, to inference a state means that the values are set by the system functions based on the inputs the system receives from the external systems (e.g. the Driver Control or the Speed Sensor). The description of the inputs and outputs is modelled separately with a different syntax within command definition sheets. Figure 8.14 contains an example template for a output command, which is similar to that of an input command, which is why we omit a description of an input command. The upper description part of the definition sheet is described in the relevant literature on SpecTRM-RT, which the reader is relegated to, at this point. More interesting is the command definition section, in which the transitions of the state variable definitions are specified. The transition definitions maintain the consistency between the state descriptions in the controller model and the transition logic that is triggered by the input / output states. The rationale behind the transition descriptions are pre- and post-conditions. Please consider the example in Figure 8.15 which show the post-condition "= *mode-TR*" that has to hold if the pre-condition specified in the sub-sequent list is met. In this example the only pre-conditions is the operation "*RemoteTrackControl.SetMode() = mode.TR*" that has to be *true* (indicated by the adjacent "*T*" in the box). The syntax for stating pre-conditions is tabular that prescribes one row for each predicate and the relevant value. Expressing predicate logic operators such as $\exists$ or $\forall$ is achieved via truth tables reflecting the relevant value combinations of the specified predicates.

Pros:

- SpecTRM-RL formalism allows backtracking of output states to their root causes in the input state space.
- Very good at modelling black box systems as state machines (e.g. embedded controllers).
- Graphical notation provides a good overview of the existing state variables. The basic concepts / components of a system can be examined very quickly.
- Good overview of the state transition behaviour of pre- and post-conditions by the use of truth tables.

Cons:

- Not suitable to handle set operations.
- The graphical notation is heavily tailored towards modelling black box systems. Not so flexible in other domains.
- Some typical mechanisms (e.g. used in Object Orientation) such as typing and derivation are missing. (e.g. if one may want to model that a break or an engine is a kind of a train trip actuator)
- The quantification of propositional expressions used in pre-/post-conditions is modelled via truth tables, which significantly raises the complexity in rules with multiple expressions.
- The behaviour is modelled via state transitions that are expressed by spreading them all over multiple document pages, which makes it difficult to follow the semantics of the overall state machine.

8.5.3 UML-Activity Diagram

The UML-Activity Diagram is a graphical notation for modelling control and data flows. It has been devised to model behaviour of software systems and is part of the UML-specification[9]. This section gives a brief overview of the notation to make the reader familiar with the basic concepts of the notation.

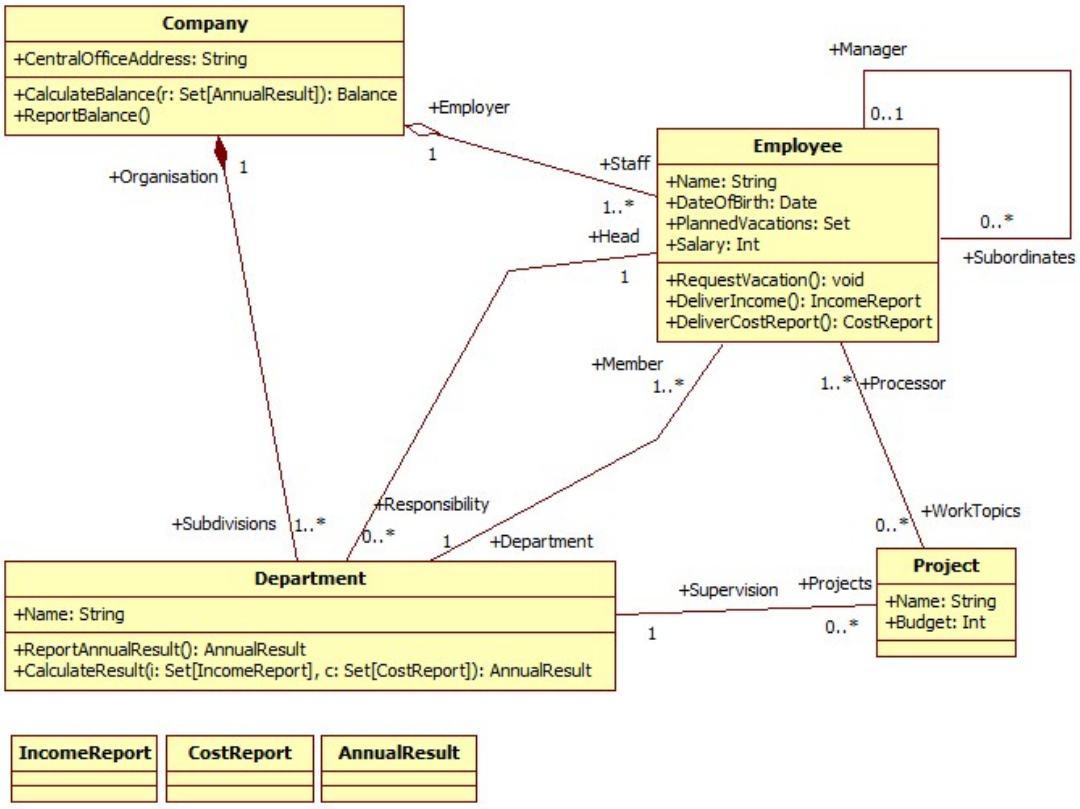


Fig. 8.16: Example Concept Model as Basis for UML-Activity Diagrams

Language constructs of UML-Activity Diagrams:

- *Action Node* (rectangle with round corners): Denotes the name of the action indicated by a meaningful expression that allows to conclude on the related class method in the UML-Class model.

- *Object / Control Flow* (arrow between nodes): Denotes the control / object handover and direction between the involved actions. Can be constrained by *Selection*

[9] in Chapter "Activities" in the UML-specification available under http://www.omg.org/spec/UML/2.5/PDF (latest)

/ Transformation Behaviour indicated by a guard condition based on a logical expression or a reference to a model concept that stands as a placeholder for that logic expression.

- *Object Node* (rectangle): Denotes an object derived from an UML-class.
- *Object Node Pins* (small rectangles between Object / Control Flow and Action Nodes): Denotes that a data flow shall be modelled between a providing and a consuming action.
- *Selection / Transformation Behaviour* (rectangle with an indicated wrinkle at the top-right corner): Selection behaviour allows to specify filter and order constraints to the data (object model) exchanged between the providing and the accepting action. Transformation behaviour requires the invocation of a transformation function that takes the original data (object model) as input and transforms it to an object model that complies with the data type of the accepting action.
- *Initial Node* (filled circle): Denotes the start of a control flow.
- *Final Node* (small filled circle with a bigger concentric unfilled circle): Denotes the termination of a control flow.
- *Merge / Decision Node* (unfilled diamond with arriving / departing arrows): A *Decision Node* allows to specify an If-Then-Else semantics to the *Control Flow* by specifying textual OCL-constraints on the leaving arrows (denoted by "["*<OCL-Constraint>*"]" adjacent to the relevant end of the arrow, leaving the decision node).
- *Fork Node* (bar with one arriving and one to many departing arrows): Denotes the parallelization of a control / data flow.
- *Join Node* (bar with two or many arriving arrows and one departing arrow): Denotes the sequencing of a control flow / merging of a data flow.

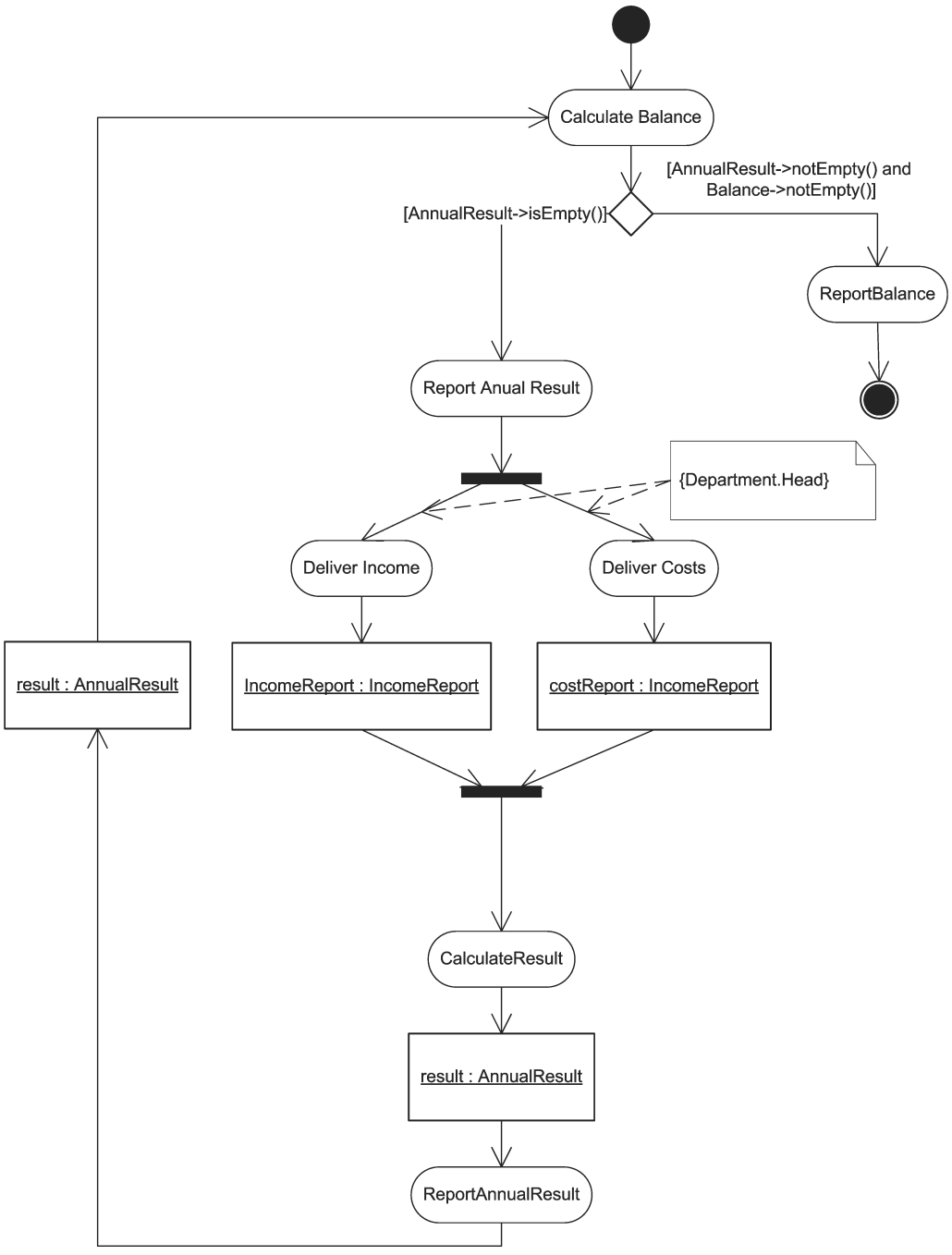


Fig. 8.17: Activity Diagram for Modelling the Behaviour According to the Model in Figure 8.16

Figure 8.17 describes a UML-Activity diagram, that specifies the workflow for generating the balance of a company (see also UML-class model in Figure 8.16). The workflow begins with invoking the method *CalculateBalance* (indicated by the Action "*Calculate Balance*") of the class *Company*. Which leads to the first decision, that says if a dataset called annual result is not available. Let us assume, that initially the criterion for this transition holds, which leads to the action "*Report Annual Result*", representing the method *ReportAnnualResult* of the class department. Method *ReportAnnualResult* leads to the invocation of the subsequent two methods *DeliverIncome* and *DeliverCosts*, both located in the *Employee* class. In this business case, this action requires the leading employees, restricted by the *Department.Head* statement, to deliver the relevant datasets, denoted by *IncomeReport* and *CostReport*. Once both datasets are available the *CalculateResult* method of class *Department* produces an *AnnualResult* dataset and passes it to the method *CalcualteBalance* over the method *ReportAnnualResult* in *Department* as a step in between. Since now the annual result exists and the balance is assumed to be calculated based on the annual result input datasets, the alternation (denoted by the diamond) takes the path over the *ReportBalance* action and the workflow terminates.

Pros:

- Quick overview of the behaviour of a process. Control flow can be modelled as a white box.
- Concise grammar, which is considered to be understood easily, also by non-programmers.
- Part of the widely accepted UML standard.

Cons:

- Not expressive enough for constraining object models compared to the possibilities of predicate logic (APRIL). Syntax for describing Rules as *Selection / Transfor-*

mation Behaviour nodes is very limited. This can only be bypassed by defining auxiliary UML-class attributes and referencing those.

- Before executing the business logic described with UML-Activity diagrams, they have to be translated into an executable target language. This could either be a common programming language like e.g. Java (which is also suitable for generating the productive implementation if the model is detailed enough) or into a set of OCL-Pre-/Post-Conditions (e.g. the Bremen USE-Tool can be used for executing). Additionally [57][10] shows how to generate test cases from UML-Activity diagrams.

- cardinalities are not made explicit (just a personal favour of the author).

8.5.4 Comparison

In this section some considerable features of the presented frameworks are briefly compared, which is listed in Table 8.3.

In this comparison, it arises that the expressive power of APRIL is suitable for business and technical domains, although with respect to the overall method it appears worthwhile to include the automated test code generation into the APRIL framework.

[10] http://www.sts.tuhh.de/pw-and-m-theses/2014/kurth14.pdf

	Alloy	**SpecTRM-RT**	**UML-Activity Diagrams**	**APRIL**
Concepts	Own language constructs to denote a concept, making small to medium size models concise and readable.	Graphical artifact to denote components and state variable values.	Textual references to concept names that exist in UML-class diagrams.	Uses UML-class diagrams, which is especially suitable for domain experts.
Relations	Syntax to denote a concept to sub-concept relationship. Allows to express links between concepts in a concise way.	Graphical Box-in-Box notation or arrows with textual IDs of state variables and constants for values.	Abilities of UML class diagrams.	Abilities of UML class diagrams.
Behaviour	Modelled as operations on discrete global states, which is not so obvious to comprehend.	Modelled as sequences of state variables and pre,-post-conditions.	Easy to understand syntax using boxes and arrows to denote control flow and data flow restrictable by gate conditions.	Possible using operators with an ITL-semantics. Or if suitable using pre, post-conditions.
Set Constraints	Very powerful syntax using special operators for specifying relation properties (e.g. reflexivity, transitivity).	not possible	No own syntax for set constraining. Only indirect via referencing already restricted class model concepts that are restricted by OCL	Possible using operators incorporated into the language.
Test Case Generation	Incorporated in the framework.	Incorporated in the framework	External tooling needed.	External tooling needed.
Test Code Generation	Statements directly executable by its related tool.	Statements directly executable by its related tool.	External tooling needed.	APRIL compiler, AnaTempura and OCL interpreter (e.g. Bremen USE-Tool) needed.

Table 8.3: Feature Comparison between Alloy, SpecTRM-RT, UML-Activity Diagrams, APRIL

8.6 Summary

In this chapter, the aim was to show that the syntax of APRIL is more understandable than that of OCL for untrained persons with little background in set theory and logic. Therefore, a case study with 30 students has been conducted who were asked to attempt to comprehend APRIL and OCL rules on different levels of complexity. The result of the study shows that it is possible for the envisaged target group to understand APRIL statements better than OCL, which is a UML standard.

Moreover, an additional case study was conducted to support the claim that APRIL is usable in real-world scenarios. Therefore, a subset of a standardised railway system specification devised by a sub-authority of the European Union has been used for the formalization as APRIL pre- and post-conditions. Using this composition also showed that pre- and post-conditions can be used to model behaviour with certain restrictions. The results indicate that from the viewpoint of expressiveness, it is possible to formalize real-world specifications. It surfaces that the restrictions imposed by the use of pre- and post-conditions for modelling behaviour forces the requirements engineer to decide on whether the APRIL rules should focus purely on domain knowledge or involve technical aspects, too. Focusing on domain knowledge neglects the introduction of synthetic state variables and methods to be attributed to the state-automaton-like specification style. This, on the one hand, enhances the understandability for human readers but, on the other hand, chops the relations between the state transitions.

The final sections of this chapter present a brief example of the overall formalization mechanism along with a comparison to other languages for specifying business rules.

Chapter 9

Conclusion and Future Work

9.1 Summary

In this thesis, a language (APRIL) is developed that allows specifying computer-interpretable business rules which are at the same time understandable to business people. The language is part of a framework which can also be used to make consistency checks and apply abstraction rules based on a novel axiomatic system based on Propositional Interval Temporal Logic. People skilled in the discipline of language design can add tailor-made atomic formulas to enhance the expressiveness of the language. So-called common constraints are by default incorporated into the language, making APRIL more expressive for certain general-purpose expressions that are rather complex to formulate in the target language, OCL.

A key aspect that caters significantly to the natural language-like readability of the formulated business rules is the mixfix notation for APRIL definition signatures. Parsing mixfix signatures in rules is evaluated by a demonstrator.

The framework also contains a methodology to apply the formalization steps from pure natural language to APRIL, enhancing requirements engineering. It has been developed during different basic case studies and represents the distillate of the steps undertaken by transforming natural language statements from the original requirements documents into the APRIL language. In addition, the methodology is extended to utilize the novel ISE method developed by Ben Moszkowski for abstracting behavioural business rules that comply with the 2-to-1 property. This part of the methodology can be regarded as an application of the ISE method in a practical context and thus helps to provide evidence that Moszkowski's method can be used in practice.

A case study conducted with 30 students trying to explore that APRIL is understandable to people with a basic background in logic and set theory by at the same time very little UML and APRIL training effort. The study also revealed that APRIL is ahead in understandability to OCL. Surprisingly the case study also surfaced that the UML-class model is particularly well to comprehend despite its semi-formal character. This circumstance supports the claim that the language mix APRIL imposes, is suitable for

people with only little APRIL and UML experience.

9.2 Contributions

The thesis proposes a novel framework to cater to the quality of requirements documents by allowing to express business rules formally using a novel language APRIL, which is based on UML-class models, OCL and Tempura. The entire requirements engineering process is supported by the APRIL-framework, comprising the language APRIL itself, a methodology as a guideline to formalize natural language business rules in APRIL, a subset of a novel technique (ISEPI) based on PITL to reduce the unintended complexity of business rules. The language APRIL allows to express statements as sentence patterns that can intriguingly mimic natural language statements. This is supported by operators that can be expressed in mixfix syntax, for which an algorithm has been devised that can resolve even nested mixfix statements in a parser. The way mixfix is used in APRIL is uncommon to programming languages.

The contributions are as follows:

- A technique and methodology to use semantically well-founded sentence patterns using mixfix notation to mimic natural language sentences, which contributes to the readability of specification documents (even partly) written in APRIL.

- A technique to incorporate domain-specific atomic formulas into the APRIL-language. The usage of compiler building mechanisms to directly change the APRIL-language and enhance the expressiveness of the language makes APRIL an extremely powerful and expressive tool.

- An overview of the key aspects of state of the art software development and how the APRIL framework can be integrated into the respective processes for enhancing requirements engineering.

- A set of general purpose common constraints that cater significantly to the

expressiveness of APRIL. The representation of common constraints in the target language OCL would be very complicated for the envisaged target group of users, hence the developed common constraints support the utilization of the language in this area.

- An investigation on the understandability of APRIL, indicating that the APRIL is indeed more understandable than OCL. This investigation also revealed that UML-class models are also sufficiently comprehensible for untrained persons even with no initial training to support the understanding of related APRIL business rule statements.

- A formal syntax and semantics of the new language APRIL to be used for invariant, pre- and post-conditions as well as behavioural business rules. The semantics is based on predicate logic and Interval Temporal Logic.

- A technique for reducing obvious semantic flaws in APRIL specifications using consistency checking based on the tools MONA and PITL2MONA. The basic idea is to transform PITL-formulas into a suitable representation of ws1s[3] using PITL2MONA. Once compiled to a checkable form, established model checking methods based on ws1s can be applied using the MONA-Checker.

- An evaluation and first quantification towards the practical usability of a subset (ISE) of Moszkowski's novel ISEPI method. ISE is used for abstracting and simplifying behavioural business rules to make their syntax more concise.

9.3 Success criteria revisited

Here is the matching of the results and the envisaged success criteria:

- *Elaboration of the criteria that a formal language has to meet to be suitable for requirements engineering*

[3]Weak monadic second-order theory of one successor

The investigation on related work (Section 2.8) and the introspection of state of the art software development and requirements engineering methods (Chapter 2) showed that a controlled natural language along with enhancing techniques appears reasonable to be used for requirements engineering. This is materializing in the APRIL framework (Chapters 4, 5 and 6).

- *Definition of the syntax and semantics of this language*

  The syntax is described by an EBNF formal grammar (Sections 4.6.1 ff). The semantics is defined as a denotational version in a tabular form that maps each atomic formula representation of the syntax to a semantic counterpart (Section 4.6.7). For invariant, pre- and post-condition business rules the target language OCL is used, whereas for behavioural business rules the Interval Temporal Logic is chosen.

- *Definition of the practical contribution and implementation in a software development process by means of a framework*

  Therefore, the APRIL framework has been devised comprising the APRIL language (Sections 4.6.1 and 4.6.7), methodology (Chapter 6), consistency checking and rule abstraction techniques (Chapter 7). Moreover, a proposal to integrate the framework into existing development processes is sketched (Section 6.2).

- *The capability of the framework to contribute an additional value in requirements engineering*

  The additional value is, on the one hand, materializing as a technique for consistency checking of behavioural business rules that is able to find contradictions in a requirements specification. On the other hand, a novel ISE technique is used to abstract business rules to find simpler universal representations of a sequence of business rule intervals that is faster to evaluate for computers and easier comprehend for humans (Chapter 7).

9.4 Future work

The proposed techniques in the APRIL framework are both individually challenging and untypical in their combination to be used in requirement engineering. Especially the 2-to-1 property and the ISE method are novel and thus not commonly known. Thus, future research can target on developing educational methods to introduce the framework to business people. Therefore, the thesis provides a basis for future research in this area. Aspects, worth investigating further can be sketched as follows:

- Elaborating a hybrid approach that uses NLP to help transform natural language requirements specifications by extracting the conceptual domain model, the semantics of the verb phrases and help to extract an unambiguous business vocabulary.

- Developing an education model, suitable for APRIL-users, comprising linguistics, formal languages, compiler construction, requirements engineering and formal methods, especially in the disciplines predicate and temporal logic as well as logic prove strategies.

- The model checker based detection of inconsistencies in business rule specifications can be enhanced in two ways.

  First, today the interpretation of counterexample results is a topic that is left to the user who has to comprehend the counterexample first by remapping the state back to the domain model and based on that, she has to decide if the counterexample is critical or even possible within the described domain. Hence, there is a need to support the remapping of the counter example back to the domain model.

  The second way could be to automatically generate a PITL-formula that describes exactly the counterexample, which then allows refining the originally checked formula by including the generated formula. By doing so an exception is modelled into the formula with very little effort and the subsequent checks iterate further

on. In this context, it could also be feasible to create model checkers that can be configured with a set of some kind of meta-rules which are based on the domain model and that formally describe uncritical states that can be omitted. This would leave the original formulas concise.

- A further enhancement for abstracting behavioural business rules can be provided by a tool support for automating the ISE method and detecting the 2-to-1 property.

- It could be worthwhile to discover if the extension of the ISE method, which according to Moszkowski [77], using parallel combining and with the Integration step becoming the fully-fledged ISEPI abstraction mechanism, can produce better results in abstracting business rules than the ISE subset. From our viewpoint, it would also be interesting to apply the ISEPI method on APRIL business rules that describe concurrent processes.

- Extending APRIL with atomic formulas requires language designer and compiler builder skills which are very likely to go beyond the skills of the typical requirements engineer. Hence, a technique or methodology could be devised which unites the steps required into a concise and easier to understand framework. E.g., each step could be supported by the following:

    - Defining new language constructs could be supported by an automated ambiguity detection that allows more easily fitting the new language constructs into the existing formal grammar.

    - To combine the syntax definition more easily with the semantics for new language constructs, an integrated technique could be conceived, guiding the user in specifying the EBNF rules along with the target-language template. For testing the template for meeting its intended semantics, an instantiation mechanism along with the runtime environment used to instantly get a response on the correctness would help to speed up the domain-

specific tailoring of APRIL and increase the acceptance of the entire APRIL-framework.

- Powerful tooling featuring state-of-the-art development environment would boost the acceptance of APRIL significantly. However, this is out of the scope of this thesis as it is more into product development, which would exceed the effort for a PhD degree. However, it would indeed be interesting to repeat the acceptance case study using a development environment to be created. Thus, the hypothesis is that in the discipline of correctly formalizing business rules into APRIL statements, the results would be significantly better.

Appendices

Appendix A

Tooling

A.1 Case Tools

A.1.1 Visual Paradigm for UML 7.0

Visual Paradigm for UML is a Case-Tool enabling to model UML 2.1 models. Code generation for Java, .Net and PHP is supported. Although there are plenty of modelling features, a straight forward engineering process supported by configurable code generation is only possible in a very restrictive manner. Pros

- Very good user interface
- Integration of model transformation mechanisms (e.g. UML-to-Entity-Relation-Model and vice versa). Additional data base integration.
- Data-export and import in XML-format.
- All UML-models in the newest version are supported
- Model element notation can be extended

Cons

- Code generation is not customizable.
- Model transformation mechanisms are not customizable

A.1.2 Borland Together

Borland Together is a case tool that allows defining BPMN Models along with UML models. Together is based on the Eclipse platform using the Eclipse-Modelling-Framework (EMF) [26]. UML modelling comes along with a mature OCL support that enables model validation. Additionally is supports the definition and processing of user-defined DSLs. Model-to-Model transformation in Together is realized via QVT [89], which makes the transformation mechanisms customizable. Code generation (also Model-to-Text transformation) is done with Java classes based on a given meta-model. The actual code output is produced by the classes predefined output commands. Moreover, a new template engine can be used to generate code in parallel, which in fact seems not fully integrated. Along with some proprietary formats of minor importance, a standardized data exchange possibility comes with an XMI-compliant import/export mechanism. Pros

- Eclipse based
- Very good documentation and tutorials
- Version control mechanisms
- Supports Round-Trip Engineering

Cons

- Weakly integrated template mechanisms for code generation

A.1.3 Magic Draw

Magic Draw created by NoMagic is available both with a proprietary stand alone user interface and as plugin for the Eclipse platform. The abilities for model creation and model validation are comparable to that of Borland Together (see A.2.4.1.5.1.2). BPMN along with UML 2.0 are supported by default, while the latter can be extended by the

UML inbound profile mechanism in order to create domain specific notations. The graphical user interface seems mature, is intuitively usable and all in all the best out of off all the tools evaluated within this document. In order to define rules upon the model, OCL is used. However, the specification of the models is done by a set of entry forms, instead of a state of the art IDE-like editor, which transports an antiquated look and feel. The core competency of Magic Draw is clearly on model creation but it also has abilities of code generation especially for platform-specific code like C++, Java, C#, CORBA and others. These abilities are mainly restricted to generating simple code bodies without implementation logic (e.g. a OCL-to-Java Compiler is missing). Model transformation abilities are limited to a non-customizable UML to Entity-Relationship-diagram transforming mechanism which can be conducted vice versa. Pros

- Wide range of mature Reverse-Engineering abilities
- Best User interface
- Eclipse based

Cons

- Intricate OCL "editor"
- No customizable code generation mechanisms
- No customizable model transformation mechanisms

A.2 MDSD Tools

A.2.1 MetaEdit+

MetaEdit+ allows creating and evolving graphical model elements which then can be used for specifying domain models. These domain models are the basis for code generation. The transformation logic is specified in code templates. It is possible to

assign multiple templates to a single model element type. Moreover, model elements of different languages can be put into a decomposition relation. For example a complete, model is one language can be represented by a single model element of another element. A significant strength of the MetaEdit+-tool is the model checking ability, which is based on rules that have been defined along with the model elements. MetaEdit+ ships with a list of debuggers (e.g. supporting a visualization of received values of a web service in the model at runtime) Pros

- Good graphical editor
- Customizable model elements
- Visualizing runtime data in the model
- Model checking mechanisms
- Good DSL creation support

Cons

- Text editors for templates have only few textual IDE-features

A.2.2 Meta Programming System

Jet Brains's Meta Programming System (MPS) is a language workbench for creating and maintaining external textual DSLs. New textual model parts are defined using the internal Java-like so called BaseLanguage, which is actually the meta-model for any MPS-based DSL. A model that is specified in an MPS-based DSL is compiled into java. In order to conveniently test new implemented DSL concepts a sandbox is provided as well. A considerable step in order to achieve to compile DSL-code is to map DSL concepts to constructs of the target language to enable to automatically generate a compiler. Pros

- Mature Editor
- High flexibility in defining external DSLs according to the meta model

- Good template language for creating the compiler
- Very good documentation

Cons

- Integration with other tools is difficult as MPS is pretty self-contained
- Only textual DSL concepts usable

A.2.3 AndroMDA

AndroMDA is a Java based open source framework without having an own editor or development environment. Roughly, its functionality is comparable to the former tools but as lacking an editor by standard it is by far less convenient and usable out-of-the-box. However, it can be integrated into the Eclipse platform. Its architecture is well structured and allows integrating extensions. Pros

- Flexible extensible (template engine, functionality)
- Works together with existing development environments.

Cons

- Complicated use and must be integrated into existing IDEs
- Standard template engine is the antiquated Velocity

A.2.4 Eclipse Modelling und open Architecture Ware

The open Architecture Ware (oAW) is the MDSD-offspring of the Eclipse platform. The core of oAW is another framework called Eclipse Modelling Framework (EMF). EMF supports the definition of Domain-Specific Languages and additionally provides an XML based persistency layer to enable to directly save DSL models into a database. With the Graphical Modelling Framework (GMF) it is possible to specify graphical DSLs.

However, the use of GMF is, compared to other tools, not as intuitive and has to partly be supported by programming interventions. On the counterpart to GMF for specifying graphical DSLs the Textual Modelling Framework (TMF) is integrated to help develop textual modelling languages. For model checking OCL rules can be defined on each language concept. Model-to-model transformations can be conducted also using QVT, ATL, or Xtend. Model-to-Code transformations are defined by templates of the template engine Xpand. A convenient feature comes with the automated generation of an IDE, which is based on the DSL specification. The modular architecture of oAW allows substituting certain components. Pros

- Good coverage of the MDSD paradigm
- Flexible
- Eclipse-based

Cons

- Poor documentation
- Conglomerate of many different potentially independent components making the configuration management a critical factor
- Only few patterns and references / examples
- Any generator requires several manual add-ons
- Impractical restarts required when attempting to use new model elements

A.2.5 ObjectIF

Microtool's ObjectIF is designed support modelling, model-transformations and code generation according to the Model Driven Architecture (MDA) paradigm. Therefore, UML model elements used to specify domain models along with the compliant

transformations, which are defined by a graphical transformation model. Code generation abilities are limited to compiling domain models specified in UML-Class and UML-Activity models into platform-specific Java-, C#-, Visual Basic .Net- and C++-Code. Generating WSDL-descriptions for web services based on BPMN-models is also supported. ObjectIF is available as standalone application and can be integrated into popular Microsoft Visual Studio and Eclipse Development Environments. Pros

- Integrates into existing IDEs
- Good round trip engineering
- Feels mature
- Interfaces for testing code using Nunit and Junit
- Supports methods of Requirements Engineering
- Versions management

Cons

- Transformations from PIM to PSM are obligatory, requiring a huge transformation model.
- Few model checking mechanisms
- Few IDE-typical support for writing code

Appendix B

Practical Demonstration of Mix-fix Signature Parsing

The following code snippets show how the mixfix-parsing can be achieved using C#, which has a syntax similar to Java.

```
1  private static void IsPathNameOrDefinitionCall(string[] tokenStream, ContextHandler context, ScopeStack
       scopeStack)
2          {
3              int position;
4              var result = TryGetPosition(tokenStream, out position);
5              var definitionCall = tokenStream;
6              if (position > -1)
7              {
8                  definitionCall = tokenStream.Take(position + 1).ToArray();
9              }
10
11             /*
12              * Alpha_0 as defined in the grammar (semanrtic annotation)
13              */
14             if (IsPathname(definitionCall, scopeStack))
15             {
16                 // => append PathElement to AST
17             }
18
19             RegularExpressionDefinition matchingExpression;
20
21             /*
22              * Alpha_1 as defined in the grammar (semanrtic annotation)
23              */
24             if (IsDefinitionCall(definitionCall, scopeStack, out matchingExpression))
25             {
26                 // => append definitin call to AST
27
28                 // search for embedded calls and index the elements in matchingExpression
29                 GetCallerIndexes(matchingExpression, definitionCall);
```

```
30              foreach(RegexVariable variable in matchingExpression.Variables)
31              {
32              ____// extract embedded definition call
33                  var embeddedPathNameOrDefinitionCall = tokenStream.Skip(variable.SymbolBoundary.Start).Take(
    variable.SymbolBoundary.Lenght).ToArray();
34
35                  // recursive call
36                  IsPathNameOrDefinitionCall(embeddedPathNameOrDefinitionCall, context, scopeStack);
37              }
38          }
39      }
```

Listing B.0.1: Mix fix parsing in C#

```
1       /// <summary>
2       /// 1. Method does not support 2 two or more subsequent calls (varaibles)
3       /// 2. Method does not support variable constituent ids that are named equally to the subsequent constant
     sequence.
4       /// => do support both a regexmatch and type check for embedded calls has to be impelented => see Hint 1
5       /// </summary>
6       /// <param name="D_regex"></param>
7       /// <param name="DefinitionCall"></param>
8       private static void GetCallerIndexes (RegularExpressionDefinition D_regex, string[] DefinitionCall) //
    Eliminates Constants
9       {
10          if (D_regex.NumberOfVaraibles == 0) return;
11
12          RegexConstant currentSynchToken = D_regex.GetNextConstant();
13          int tokenIndex = -1;
14          int definitionCallLenght = DefinitionCall.Count();
15
16          foreach (RegexElement element in D_regex.Expression)
17          {
18              tokenIndex++;
19              var variable = element as RegexVariable;
20              if (variable != null)
21              {
22                  variable.SymbolBoundary.Start = tokenIndex;
23                  int i = 0;
24                  string combinedTokens = string.Empty;
25
26                  do
27                  {
28                      // HINT 1:
29                      // in order to make CVVC signatures work it is necessary to impelement a recursive
    patternmatching here including a type check.
30                      // build signature one by one and check against the definitions (D*_regex) in the scope
    of the resuolution step
31                      // problem is that embedded calls may not use equal synch tokens.
32                      if (D_regex.EndOfConstants)
33                      {
34                          variable.SymbolBoundary.End = definitionCallLenght - 1;
35                          break;
36                      }
```

```
37
38                      // HINT 2: the synchtoken (the constant that termintes the token tream of the varaible) is folded accordingly to its lenght.
39                      //         this softens the broblem of HINT 1, which says that variable constituents may not contain tokens that are equal to their preceeding (single) synchtokens.
40                      //            Now the problem can be bypassed by using multiple constants folded together to one.
41                      combinedTokens = string.Empty;
42                      for (int combIndex = 0; combIndex < currentSynchToken.Lenght; combIndex++)
43                      {
44                          combinedTokens += DefinitionCall[i + tokenIndex + combIndex];
45                      }
46
47                      if (currentSynchToken != null && combinedTokens == currentSynchToken.Value)
48                      {
49                          tokenIndex += i - 1;
50                          variable.SymbolBoundary.End = tokenIndex;
51                          break;
52                      }
53                      i++;
54                  } while (combinedTokens != currentSynchToken.Value || !D_regex.EndOfConstants);
55              }
56              else if (element is RegexConstant)
57              {
58                  // add the lenght of the previous synchtoken (consider that one has lready been added so, -1)
59                  if (currentSynchToken != null)
60                  {
61                      tokenIndex += currentSynchToken.Lenght - 1;
62                  }
63
64                  currentSynchToken = D_regex.GetNextConstant();
65              }
66          }
67      }
```

Listing B.0.2: Synch token detection for mix fix parsing in C#

```
68
69  internal class CallSymbolBoundary
70  {
71      public int Start { get; set; }
72      public int End { get; set; }
73      public int Lenght { get { return End - Start + 1; } }
74  }
75
76
77      internal class RegularExpressionDefinition
78  {
79      private bool m_EndOfConstants;
80      private int m_currentConstantIndex = 0;
81      private int m_cosntantArrayLeght;
82      private RegexConstant[] m_constantArray;
83
84      public RegularExpressionDefinition(IEnumerable<RegexElement> elementsOfExpression)
```

```
85          {
86              Expression = elementsOfExpression;
87              m_constantArray = elementsOfExpression.OfType<RegexConstant>().ToArray();
88              m_cosntantArrayLeght = m_constantArray.Length;
89          }
90
91          public IEnumerable<RegexElement> Expression { get; private set; }
92
93          public IEnumerable<RegexVariable> Variables
94          { get
95              {
96                  return Expression.OfType<RegexVariable>();
97              }
98          }
99
100         public bool EndOfConstants
101         {
102             get
103             {
104                 return m_EndOfConstants;
105             }
106         }
107
108         public int NumberOfVaraibles
109         {
110             get
111             {
112                 int i = 0;
113                 foreach (RegexElement element in Expression)
114                 {
115                     if (element is RegexVariable)
116                     {
117                         i++;
118                     }
119                 }
120                 return i;
121             }
122         }
123
124         public RegexConstant GetNextConstant()
125         {
126             if (m_cosntantArrayLeght > m_currentConstantIndex)
127             {
128                 var constant = m_constantArray[m_currentConstantIndex];
129                 m_currentConstantIndex++;
130                 return constant;
131             }
132             m_EndOfConstants = true;
133             return null;
134         }
135
136         public bool Match(string stringToMatch)
137         {
138             Regex systemRegex = ConstructSystemRegex();
139             return systemRegex.IsMatch(stringToMatch);
140         }
141
142         private Regex ConstructSystemRegex()
```

```
143         {
144             string pattern = string.Empty;
145             foreach(var element in Expression)
146             {
147                 if (element is RegexConstant)
148                 {
149                     pattern += element.Value;
150                 }
151                 else if (element is RegexVariable)
152                 {
153                     pattern += ".+"; // .+ is .net wildcard for one or many characters
154                 }
155                 else
156                 {
157                     throw new InvalidOperationException("unknown element type");
158                 }
159             }
160             return new Regex(pattern /*+= "*"*/); // if there is a rest the * wildcard is used to match from the
    start optimistic => refactor in a seperade mode
161         }
162     }
163 
164         private class RegexElement
165         {
166            public RegexElement(string value)
167             {
168                 Value = value;
169             }
170 
171            public string Value { get; set; }
172         }
173 
174         private class RegexConstant : RegexElement
175         {
176             public RegexConstant(string value) : base(value)
177             {
178             }
179         }
180 
181         private class RegexVariable : RegexElement
182         {
183             public RegexVariable(string value) : base(value)
184             {
185             }
186 
187             CallSymbolBoundary m_SymbolBoundary;
188 
189             public CallSymbolBoundary SymbolBoundary
190             {
191                 get
192                 {
193                     if(m_SymbolBoundary == null)
194                     {
195                         m_SymbolBoundary = new CallSymbolBoundary();
196                     }
197                     return m_SymbolBoundary;
198                 }
199             }
```

```
200         }
```

Listing B.0.3: Auxiliary types and methods for mix fix parsing in C# (Part 1)

```
201         /// <summary>
202         /// According to semantic annotation, alpha_0
203         /// </summary>
204         /// <param name="PathNameOrDefinitionCall"></param>
205         /// <param name="visibleEntities"></param>
206         /// <returns></returns>
207         private static bool IsPathname(string[] PathNameOrDefinitionCall, ScopeStack visibleEntities)
208         {
209             return IsInContext<DefinitionLocalModelRelated>(PathNameOrDefinitionCall, visibleEntities);
210         }
211
212         private static bool IsDefinitionCall(string[] PathNameOrDefinitionCall, ScopeStack visibleEntities, out
        RegularExpressionDefinition matchingExpression )
213         {
214             if(IsInContext<AllDefinitions>(PathNameOrDefinitionCall, visibleEntities))
215             {
216                 matchingExpression = (RegularExpressionDefinition)visibleEntities.Contexts.Single(t => t is
        AllDefinitions).GetMatchingExpression<RegularExpressionDefinition>(PathNameOrDefinitionCall);
217                 return true;
218             }
219             matchingExpression = null;
220             return false;
221         }
222
223         private static bool IsInContext<T>(string[] pathNameOrDefinitionCall, ScopeStack visibleEntities)
224         {
225             ContextBase matchingContext;
226             if (visibleEntities.IsMatch(pathNameOrDefinitionCall, out matchingContext))
227             {
228                 return matchingContext is T;
229             }
230
231             return false;
232         }
```

Listing B.0.4: Auxiliary types and methods for mix fix parsing in C# (Part 2)

```
233 internal class AllDefinitions : ContextBase
234     {
235         private List<RegularExpressionDefinition> m_RegularExpressionDefinitions;
236
237         public AllDefinitions(IEnumerable<ITypedSymbol> typeSymbols) : base(typeSymbols)
238         {
239             m_RegularExpressionDefinitions = new List<RegularExpressionDefinition>();
240         }
241
```

```
242         public override bool IsMatch(IEnumerable<string> tokenStream)
243         {
244             foreach(var expression in m_RegularExpressionDefinitions)
245             {
246                 if(expression.Match(String.Join("",tokenStream)))
247                 {
248 
249                     return true;
250                 }
251             }
252 
253             return false;
254         }
255 
256         public override object GetMatchingExpression<T>(IEnumerable<string> tokenStream)
257         {
258             foreach (var expression in m_RegularExpressionDefinitions)
259             {
260                 if (expression.Match(String.Join("", tokenStream)))
261                 {
262 
263                     return expression;
264                 }
265             }
266 
267             return null;
268         }
269 
270         public override void AddToContext<T>(T expression)
271         {
272             var exDef = expression as RegularExpressionDefinition;
273             if (exDef == null) throw new InvalidOperationException();
274 
275             m_RegularExpressionDefinitions.Add(exDef);
276         }
277     }
```

Listing B.0.5: Auxiliary types and methods for mix fix parsing in C# (Part 3)

```
278     public interface IType
279     {
280 
281     }
282 
283         public interface ITypeContainer
284     {
285         IEnumerable<ITypedSymbol> AllClasses { get; }
286     }
287 
288         public interface ITypeDescription
289     {
290         IEnumerable<ITypedSymbol> NavigableTypes { get; }
291 
292         string Symbol { get; }
```

```
293
294         IType Instances { get; }
295     }
296
297         public interface ITypedSymbol
298     {
299        string Symbol { get; }
300
301        ITypeDescription Type { get; }
302
303        IEnumerable<ITypeDescription> Types { get; }
304     }
305
306         public class TypeSymbol : ITypedSymbol
307     {
308         public TypeSymbol(string symbol, ITypeDescription type)
309         {
310             Symbol = symbol;
311             Type = type;
312         }
313
314         public TypeSymbol(string symbol, IEnumerable<ITypeDescription> multipleType)
315         {
316             Symbol = symbol;
317             Types = multipleType;
318         }
319
320         public string Symbol
321         {
322             get; private set;
323         }
324
325         public IEnumerable<ITypeDescription> Types
326         {
327             get; private set;
328         }
329
330         public ITypeDescription Type
331         {
332             get; private set;
333         }
334     }
```

Listing B.0.6: Auxiliary types and methods for mix fix parsing in C# (Part 4)

Appendix C

Object Test Language

C.1 APRIL Object Test Language

This section presents the specification of the APRIL Object Test Language.

C.1.1 Introduction

APRIL is a declarative language to bring together natural language and predicate-logic. This is to enable business people to comprehend business rules with a formal background. APRIL requires a discourse universe, which is constructed by a UML-class model.

C.1.2 Situation

In APRIL decomposition is a proper means to reduce the complexity of certain large business rules into smaller, better comprehensible fragments. Here, so called APRIL-Definition encapsulate parts of the specified logic. An interface to inject input data into the Definitions is provided by an optional parameter list, a well known mechanism from popular programming languages. They also have a return value which is in the simplest case a truth value true, false and never a void type. However, when dealing with very complex business rules the borders to imperative implementation logic are exceeded very quickly (see Section B.4.5). In these peculiar situations imperative languages are better suitable then declarative languages. Therefore, APRIL allows specifying a comment

in the body of a Definition, which serves as help for the developer implementing the logic. However, these comments have to be in unstructured natural language and will not be interpreted. Hence, APRIL provides a test specification language (OTL), which is specified along with the comment. The idea behind OTL is to first specify an object model containing the actual test data on which, the test specification can be based on. A test specification is designed as a typical module test, providing instance data and the expected output data. Based on a test specification unit-tests in different languages can then be generated (e.g. JUnit [32]).

C.1.3 APRIL-OTL Specification

In the following the language design of OTL is presented in detail. APRIL-OTL is divided into two parts. The first part is the specification notation for the test values in the form of an object modelling language. The second part is the notation for the actual test language using the object models to denote input and expectancy data.

C.1.3.1 Object Modelling Language

The description of objects is achieved in a special syntax exclusively elaborated for this purpose. The notation for specifying objects complies with the attribute-notation. In general APRIL-OTL wraps objects by rounded brackets. The values of the object attributes are assigned by a key-value-pair (e.g. (key1="value1", . . . , keyN="valueN")) A bolder example (see Listing C.1.1) based on an example domain model (see Figure C.1) can be given as follows:

```
1 ( surName="John", lastName="Dunbar" , title="Mr.", isMale=true, dateOfBirth= "01.12.1970", customerIsPremium=
      false )
```

Listing C.1.1: OTL Example of Anonymous Objects

The upper statement defines an anonymous object of type Customer, which has no links to other Customer-objects. The benefit of that notation style is that attributes can be

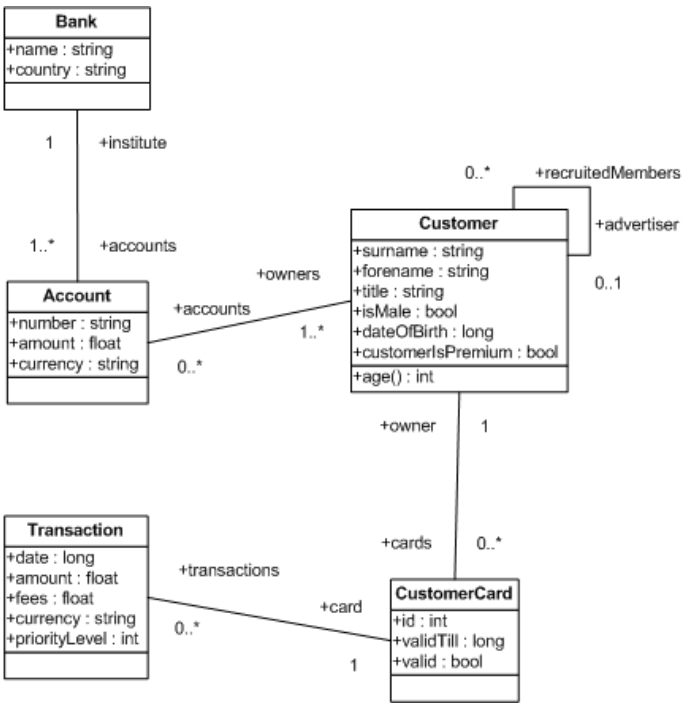


Fig. C.1: Bank Customer Example

addressed distinctively, which is convenient especially if only few attributes have to be given a value. Object-attributes to remain without a value can be left out.

C.1.3.1.1 Named Objects Named objects are introduced by an identifier and can thus be referenced as a value. The specification of named objects is always followed by a type descriptor, which is the class name of the related UML-class model. Please follow the example in Listing C.1.2.

```
1 Customer
2
3     jonni ( surName="John" , lastName="Dunbar" , title="Mr." , isMale=true
4                     , dateOfBirth="01.12.1970", customerIsPremium=false )
5
6     kathi ( surName="Kathleen", lastName="McDonald" , title="Mrs." ,
7             isMale=true, dateOfBirth="11.12.1970", customerIsPremium=true )
```

Listing C.1.2: OTL Example of Named Objects

C.1.3.1.2 Anonymous Objects Anonymous objects have no introducing identifier and cannot be referenced. In contrast to the named objects anonymous objects to not have to comply to those structuring conventions and can be defined in-line, which makes them pretty flexible. For example it is possible to nest objects as values of attributes of other objects. Consider the example in Listing C.1.3.

```
1 <Type>
2 <objectName1> (key1="String1", key2=(keyA="String2", keyB=true, keyC=123)) : <CastedType>
```

Listing C.1.3: Nesting of Objects

The value of the attribute with name key2 is an anonymous object introduced by an opening round bracket followed by a list of key-value-pairs according to the language specification. The " : <CastedType>" appendix of this example denotes a type cast, which may only come to carry with a UML-Generalization relation between the <Type> (as super type) and <CastedType> (as sub type) class.

C.1.3.1.3 List Types APRIL-OTL uses the generic list types of UML, which are Set, Sequence, Bag and OrderedSet. They can be distinguished as follows: " Bag: can contain identical objects. The list is unordered " Sequence: like Bag but list is ordered " Set: does not contain identical objects (only applicable to named obejcts). List is unordered. " OrderedSet: Like Set but list is ordered.

C.1.3.1.4 Named and Anonymous Objects as List Types List types are denoted by a curly brackets wrapper. The elements of list types can either be primitive types (like integer-, float- or string-literals) or objects or other lists. Anonymous Objects as List Types The following example specifies an anonymous list object of type Sequence(Integer): 1,2,3,4,5,6,7,8,9 An example on a real life domain model can be examined in Listing C.1.4. In this example another type cast is shown denoted by : Set(Cusomer) at the end. The type of a list is by standard Bag(<Type>).

```
{
  (surName="Joe", lastName="Miller" , title="Mr." , cards={CC1, CC2} ),
  (surName="Kathleen", lastName="Smith" , title="Mrs." , cards={CC3} )
} :Set(Customer)
\end{lstlstlisting}
%Listing 21 Anonymous List Object with Link Example
%Named Objects as List Types
In analogy to the previously introduced named object list objects are also clustered by type. The type
      declaratory is heading the specification group accordingly. The following Listing \ref{
      otl_named_list_object_example} is given as exemplification. Additionally the rule for denoting an empty list
       is given by curly brackets with an empty string in between.

\begin{lstlisting}[caption={Named List Object Example}, label={otl_named_list_object_example}]
Set(Customer)
 SetCust1
{
  ( surName="Joe", lastName="Miller" , title="Mr." , cards={CC1, CC2} )
,
  (surName="Kathleen", lastName="Smith" , title="Mrs." , cards={CC3} )
}
SetCust2 {}
```

Listing C.1.4: Anonymous List Object with Link Example

C.1.3.1.5 Linked Objects Objects are linked according to the associations of their related classes. Typically, when associated classes get implemented by object-oriented languages like Java, for each associated class-counterpart an attribute of the according type gets generated. The name of that attribute is determined by the role name of the opposing association end. Another aspect of the determination of the type of the attribute is the multiplicity of that role. APRIL-OTL complies with that schema. The example in Listing 23 shows two linked classes.

```
Customer
  kathi ( surName="Kathleen" , lastName="McDonald" , title="Mrs.")
  jonni ( surName="John" , lastName="Dunbar" , title="Mr.",
                                          recruitedMembers={ kathi })
```

Listing C.1.5: Linked Objects Example

By the use of the role name with multiplicity 0-to-many the attribute recruitedMembers introduces a link list that contains identifiers of potentially 0 to many objects.

C.1.3.1.6 Linking Rules concerning Multiplicities

Specifiying objects models based on class models have to respect several rules with respect to multiplicities. " 1 or 0-1: a single object or null "()" is expected. " * or 1..*: a list objet is expected. An empty object denoted by "" is allowed. Linking objects is not mandatory. Thus, if an object is not in charge of being linked to other objects the respective attribute must be left out.

C.1.3.1.7 Example Object

The following Table 8 the attribute names of a Customer - Object are shown. Object with attributes according to UML-Class surName lastName title isMale dateOfBirth customerIsPremium accounts :set(Account) advertiser : Customer cards :set(CustomerCard) recruitedMembers :set(Customer) Table 8 Customer Object according to implementation

C.1.3.1.8 Example for a Complete Object Specification

The following Listing 24 is semantically equivalent to the UML compliant object model of Figure 13.

```
1  Customer
2  __C1 ( surName="Joe", lastName="Miller" , title="Mr." , isMale=true ,
3            dateOfBirth="12.12.1966" , customerIsPremium=true, cards={CC1 , CC2} )
4
5  __C2 ( surName="Kathleen", lastName="Smith" , title="Mrs." , isMale=false,
6            dateOfBirth="3.4.1971", customerIsPremium=false, cards={CC3} )
7
8  CustomerCard
9  __CC1 ( id=8732, validTill=01042009, priorityLevel=2 , valid=false, transactions={T1} )
10 __CC2 ( id=4873598, validTill=10102015, priorityLevel=4 , valid=true, transactions={T2, T3} )
11 __CC3 ( id=9823498, validTill=20122019, priorityLevel=4 , valid=true, transactions={T4} )
12
13 Transaction
14 __T1 ( date=30062008, amount=300.0  , fees=0.0   , currency="DOLLAR" )
15 __T2 ( date=30042010, amount=1000.0 , fees=0.0   , currency="DOLLAR" )
16 __T3 ( date=06062009, amount=2000.0 , fees=30.05 , currency="EURO" )
17 __T4 ( date=06072009, amount=400.0  , fees=0.0   , currency="DOLLAR" )
```

Listing C.1.6: Textual object model according to Bank-Customer-Model

Remember: The group of object specifications is not a matter of an object set but of a group of single objects.

Figure 13 Object model according to Bank-Customer-Model

C.1.3.1.9 Object Models It is possible to address an object model by its name in order to reference a group of objects from other object models. This is motivated by the idea of reusability. The header to introduce an object model is as follows in Listing 25: Object Model <modelName> is defined as <ObjectDescriptions> Listing 25 Header of object models Objects are then defined in the ObjectDescription-body accustomed to the upper examples e.g. Listing 24. Object models are defined outside the APRIL-Constraints and Definitions.

C.1.3.1.10 Summary There are three different types of notation to define objects to be used in tests: " In line: Objects are anonymously defined in line at the valid position in the test. " Local reference: at the end of the Test description the keyword "based on" introduces an object model to define the test-data. " Referencing an external object model: After the keyword "based on" an identifier of an already existing object model is stated. It is possible to mix the three notation types. Listing 30 shows the grammar (in EBNF for XML) of APRIL-OTL.

C.1.3.2 Tests

Tests are named entities, which specify a tuple of input data and the expected output data. They are similar to the commonly used module tests of existing unit test frameworks. The benefit of APRIL-OTL is to be platform independent. Tests in APRIL-OTL can be defined against Definitions, which usually have a list of parameters for injecting certain input data at runtime. Thus, test-input-data is defined according to the type-sequence of the Definition's parameter list. A value delimiter is given by a comma (","). The example tests in Listing C.1.7 shows a test on a Definition with a sequence of parameters with the types Transaction and Customer . The type of the return value of the test is identical to the return type of the Definition.

```
1 test testX with
2  input T1 , C2 yields { (customerIsPremium=true) }
3 test testY with
4 input T2 , C2 yields { ( surName="Kathleen", lastName="Smith" cards={CC3},
```

```
5                       ( surName="Joe" ) }
6 test testZ with
7  input T3 , C4 yields {}
```

Listing C.1.7: Test Example

C.1.3.2.1 Test Scenarios Accordingly to the object models also groups of test can be defined externally addressed by a name. This is then called a Test Scenario. Due to the fact that any test is specifies against an interface all the tests in a test scenario have to be homomorphic. That means the type sequences of the input- and output- data tuples must be identical. A Test Scenario is introduced according to the example of Listing C.1.8:

```
1 Test Scenario <scenarioName> with signature <parameterTypDeclaratorList> yielding <returnType>
2   is defined as
3 <Tests>
```

Listing C.1.8: Test Scenario Example

The placeholder parameterTypDeclaratorList stands for a comma-separated list of parameter types, compliant to the targeted APRIL-Definition. The Test-body is the container for the actual tests.

C.1.4 Further Points of Potential Interest

The following list of improvements to APRIL-OTL may be characterized as optional enhancements. " Using of error types as a result value of a test if that test fails. This may simplify the specification of a test evaluation. " Using scripting for the Object Models and Test Scenarios. This may be done with the abilities provided by APRIL itself (e.g. at least one element in result satisfies that <boolExpression>)

C.1.4.1 Use Case Example

This section shows a real life use case in the domain of shift planning. As starting point for the use case please consider the following business rule in Listing 28. Preceding shift elements En and En+1 have to be clustered within the same set of shift elements S_k if the time in between E_n and E_{n+1}, which is defined as Tn_{n+1}, is smaller or equal to 10 minutes. Tn_{n+1} is evaluated as the time difference between their property values PEn+1 and PEn., both holding a time stamp. If Tn_{n+1} is greater than 10 minutes E_{n+1} ought to be the first element of a new set of shift elements S_{k+1}

Auxiliary conditions: " n and k are both natural numbers " PEn+1 > PEn Listing 28 Shift Planning Use Case Moreover, the use case considers the UML-class model in Figure 14.

Figure 14 Shift Planning Domain Model The example code in Listing 29 is based on the domain model in Figure 14. It is an example of an APRIL-Definition embedding a test in specified in APRIL-OTL. It is not the show the implementation logic but describes a test against which the implementation can be tested. external Value cluster of (shiftelements as Sequence(ShiftElement)) yielding Sequence(Secquence(ShiftElement)) is described by // ... business logic (see upper description)...

can be tested with

/* Test Scenario reference: Scenario and Definition signature must be equal */

/* Test Definition */

```
1 test test1 with
2 __input SchichtElementSequence1 yields ResultMultiset1
3 __input SchichtElementSequence2 yields ResultMultiset2
4 __input SchichtElementSequence3 yields ResultMultiset3
```

---

based on

```
1 Sequence(ShiftElement)
2 SchichtElementSequence1
3 { (shiftBegin=0700, shiftEnd=0730) ,
4   (shiftBegin=0732, shiftEnd=0800) , /*gap here*/
5   (shiftBegin=0815, shiftEnd=0900) ,
```

```
6    (shiftBegin=0909, shiftEnd=1000) }
7
8  SchichtElementSequence2
9  { (shiftBegin=0700, shiftEnd=0730) ,
10   (shiftBegin=0732, shiftEnd=0800) ,
11   (shiftBegin=0809, shiftEnd=0900) ,
12   (shiftBegin=0909, shiftEnd=1000) } /*no gap*/
13
14 SchichtElementSequence3
15 { (shiftBegin=0700, shiftEnd=0730) ,
16   (shiftBegin=0732, shiftEnd=0800) ,
17   (shiftBegin=0815, shiftEnd=0900) , /*gap here*/
18   (shiftBegin=0920, shiftEnd=1000) }
19
20 Sequence(Sequence(ShiftElement))
21 ResultMultiset1
22 { { (shiftBegin=0700, shiftEnd=0730) , (shiftBegin=0732, shiftEnd=0800) },
23   { (shiftBegin=0815, shiftEnd=0900) , (shiftBegin=0909, shiftEnd=1000) } }
24
25 ResultMultiset2
26 { { (shiftBegin=0700, shiftEnd=0730) , (shiftBegin=0732, shiftEnd=0800) },
27   { (shiftBegin=0809, shiftEnd=0900) , (shiftBegin=0909, shiftEnd=1000) } }
28
29 ResultMultiset3
30 { { (shiftBegin=0700, shiftEnd=0730) , (shiftBegin=0732, shiftEnd=0800) ,
31     (shiftBegin=0815, shiftEnd=0900) } ,
32   { (shiftBegin=0920, shiftEnd=1000) } }
```

Listing C.1.9: APRIL-OTL Code For the Shift Planning Use Case

C.2 Grammar of APRIL-OTL

In the following Listing C.2.10 the grammar of APRIL-OTL can be seen. As mentioned the OTL language incorporates into the APRIL.

```
1
2  (*
3  *  TITLE: APRIL OTL (Object Test Language)
4  *  AUTOR: CHRISTIAN ROETTENBACHER ne BACHERLER
5  *  VERSION : 0.1
6  *  DATE: 10.1.2011
7  *)
8
9  OTLModel =
10 TestDefinitionBlock
11
12  ['based on' {ObjectDefinitionList} ]
13  { (TestScenario | ObjectModel) };
14
```

```
15 AttributeDef = ID [':' GenericType] ;
16
17 ObjectModel =
18 __'Object Model' ID 'is defined as' {ObjectDefinitionList};
19
20 TestDefinitionBlock =
21  'can be tested with' TestSequenceOrReference;
22
23 TestScenario =
24 __'Test Scenario' ID [InterfaceDefinition] 'is defined as'
25 __TestSequenceOrReference ;
26
27  InterfaceDefinition =
28  _'with signature' GenericTypeList 'yielding' GenericType;
29
30 ObjectDefinitionList =
31    'object list' GenericType Object {Object} ;
32
33 Object = ID AnonymousObject ;
34
35 AnonymousObject =
36   '(' AttributeList ')' [':' ModelObject] ['is linked to' '(' ObjectLinkList ')' ];
37 ObjectSet =
38  {'{' (AnonymousObject | BasicValue) (',' (AnonymousObject | BasicValue) } '}' | '{}') [':' SetTypeDeclarator ] ;
39
40 ObjectLinkList = DedicatedObjectLinkList ;
41
42 DedicatedObjectLinkList =
43  AttributeNameObjectPair {',' attribute+=AttributeNameObjectPair};
44
45 GenericTypeList = GenericType {',' GenericType} ;
46
47 AttributeList = QualifiedAttributeList | ConstructorList ;
48
49 QualifiedAttributeList =
50 __AttributeNameValuePair {',' AttributeNameValuePair};
51
52 ConstructorList = Value {',' attribute+=Value } ;
53
54 ScenarioReferenceList = TestScenario {',' TestScenario} ;
55
56 AttributeNameObjectPair = AttributeDef '->' Object ;
57
58 AttributeNameValuePair = AttributeDef '=' Value;
59
60 SetTypeDeclarator =
61 __("Set" | "OrderedSet" | "Bag" | "Sequence" ) '(' GenericType ')';
62
63 GenericType =
64 __ModelObject | SetTypeDeclarator  | "String" | "Integer" | "Boolean" | __TupleType;
65
66 TupleType = 'Tuple' '{' GenericType '}';
67
68 TestSequenceOrReference =
69 __ScenarioReferenceList ( TestSequence ) { TestSequence } |
70 __(TestSequence) { TestSequence };
71
72 TestSequence =
```

```
73 __'test' ID 'with' Test+ ['using' __ObjectModelReferenceList ] ;
74
75 Test = 'input ' [ParameterList] 'yields' Value;
76
77 ParameterList =
78 __ParameterDef '=' Value {',' ParameterDef '=' Value} | Value {','  __Value};
79
80 Value = BasicValue | AnonymousObject | Object | ObjectSet;
81
82 BasicValue = STRING |  INT | BOOLEAN | "null" | "undefined" | FLOAT;
83
84 FLOAT = INT '.' INT;
85
86 BOOLEAN = "true" | "false";
87
88 STRING = ( "_" | "a".."z" | "A".."Z" | INT ) { ( "_" | "a".."z" | "A".."Z" | INT ) } ;
89 INT = ( "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0" ) { ( "1" | "2" | "3" | "4" | "5" | "6" | "7" |
        "8" | "9" | "0" ) };
90
91 ID = ( "_" | "a".."z" | "A".."Z" ) { ( "_" | "a".."z" | "A".."Z" | INT ) };
92
93
94 (* References are formatted in italic. Multiple ID assignment to non-terminals only for better comprehendability
       for the Grammar-Documentation. *)
95
96 Object = ID ;
97 ObjectModelReferenceList = ID ;
98 ModelObject = ID ; (* Reference to a UML-Class name*)
99 ParameterDef = ID ; (* Reference to a UML-Class method parameter name*)
100 AttributeDef = ID ; (* Reference to a UML-Class Attribute name*)
101 TestScenario = ID ;
```

Listing C.2.10: Grammar of APRIL-OTL in EBNF for XML

Appendix D

Use Case "Shift Planning"

Due to elaborate the requirements of a business rules language a case study comprising real life specification documents (SPEC) has been conducted. Amongst them was a SPEC of a large German enterprise in the logistics sector, dealing with shift planning. The first step was to draw a clearer picture of the possibilities of the language APRIL. This was gained by translating the business rule in Listing D.0.1. The result can be examined in Listing D.0.2, which shows that although the actual business rule in line two may be pretty clear, a tremendous preparation effort had been taken. The preparation comprises of two mixfix-operator Definitions, each containing additional local variables, to just prepare the data structures on which the rule logic is applied. However, the contemplated business rule is representative for a list of similar rules in the SPEC and it took a considerable amount of time to get it going. As a matter of the author's interest the business rule was implemented in Java and voila: the same result was achieved in one-third of the time taken for the APRIL-rule. However, the formulation of the APRIL and OCL versions took equal time. Listing D.0.3) gives an impression of the OCL business rule, revealing some obstacles in terms of clearness. The following business rule in Listing D.0.1 is based on the domain model in Figure D.1.

```
1 The max length of uninterrupted driving time on a train may by law not exceed a time span of 4 hours.
2 The sum of the durations of all services inside a block yields the uninterrupted driving time on a train for that
        block. This sum must not exceed 4h.
3 Conditions
4 The service has a service type which counts as driving time on a train (e.g. steering, instructing).
5 Every block consists of services on the same train.
```

```
The temporal distance between two services that are inside the same block must not exceed 10 minutes.
The temporal distance between two blocks must be greater than 10 minutes.
Services must not intersect.
```

Listing D.0.1: Real Life Business Rule: Domain Shift Planning

```
Invariant servicesOnVehicle concerns Vehicle:
    every sequence in uninterrupted driving time sequences satisfies that ( this sequence lasts not longer than
     240 minutes
     and
    every serviceList in sequence satisfies that at least one service in serviceList satisfies that
                                                                service.contractKindService.isRoadService ) .

Definition this (sequence as OrderedSet of type Service) lasts not longer than (maxTimeSpan as Integer) minutes
     is defined as
    LastElement.shiftEnd - FirstElement.shiftBegin < maxTimeSpan
with
 LastElement is defined as last element in sequence ,
 FirstElement is defined as first element in sequence .


Definition uninterrupted driving time sequences yielding Orderedset of type OrderedSet of type Service is
     defined as
    iteration over each element currentService in DrivingService yielding resultValue as OrderedSet of type
     OrderedSet of type Service beginning with StartValue
      is defined as

       if it is the case that
            last element in DrivingServices <> currentService
              and
            currentService.shiftEnd - currentService2.shiftBegin < 10
       then
            result including last element in result followed by currentService2
       else
            result followed by new OrderedSet having currentService2
       end if

   with
      currentPosition is defined as element position of currentService in DrivingServices,
      currentElement2 is defined as element at currentPosition in DrivingServices2

    end iteration //
with
UnsortedDrivingServices is defined as each service where isSteering=true and isInstructing=true,
DrivingServices is defined as UnsortedDrivingServices sorted by shiftEnd,
DrivingServices2 is defined as all elements of DrivingServices without first element in DrivingServices,
StartValue is defined as new OrderedSet having new OrderedSet having first element in DrivingServices .

```

Listing D.0.2: In APRIL Implemented Logic of Real Life Business Rule: Shift Planning

```
context Vehicle inv servicesOnVehicle:

-- filtering for services concerning driving time on the train:
let DrivingServices : OrderedSet(Service) =
 services->select(isSteering=true and isInstructing=true)->sortedBy(shiftEnd) in

 -- Construction of an auxiliary set without the first object supporting comparisons of the form:
 -- Index  1   2   3   4   5   6
 --    M1 {A , B , C , D , E , F }::OrderedSet(<type>)
 --    M2 {B , C , D , E , F     }::OrderedSet(<type>)
 --    IMPORTANT: the type OrderedSet or Sequence has to be used for both!
let DrivingServices2: OrderedSet(Service) =
  DrivingServices->excluding(DrivingServices->first()) in

-- Calculation of the latest uninterrupted service-set: BR: "Interruption shorter than < 10 min are skipped"
let UninterruptedSequences : OrderedSet(OrderedSet(Service)) =

-- Iteration of the base set; Comparison of the set elements with equal position indexes.
__DrivingServices->iterate(currentService : Service;
__    result:OrderedSet(OrderedSet(Service))=OrderedSet{OrderedSet{DrivingServices->first()}} |

-- Definition of the 2. Iterator-variable from the set (M2:=DrivingServices2)
__ let currentService2 : Service =
__     DrivingServices2->at(DrivingServices->indexOf(currentService)) in

__ if
__     -- if the current element is not the last element
____ DrivingServices->last() <> currentService

____ and

____ -- calculate comparison value: ShiftEnd(element) - ShiftBegin(element-1) < 10; BR: "Interruption < 10 min
      are skipped"
____ currentService.shiftEnd - currentService2.shiftBegin < 10

__ then

__   -- if the difference is smaller than the given value of the BR then, the service element belongs to the
      current block.
____ result->including(result->last()->append(currentService2))

__ else
__     result->append(OrderedSet{currentService2})

__ endif
__ )

in

__-- End of the last service minus beginning of the first service yields the duration of the entire sequence.
__-- BR:duration of the entire sequence (for services concerning driving time on the train) may not exceed 4 h
__ UninterruptedSequences->forAll(sequence | sequence->last().shiftEnd - sequence->first().shiftBegin < 240 and
__    sequence->forAll( services | services->exists(service:Service | service.contractKindService.isRoadService )
      ))
```

Listing D.0.3: In OCL Implemented Logic of Real Life Business Rule: Shift Planning

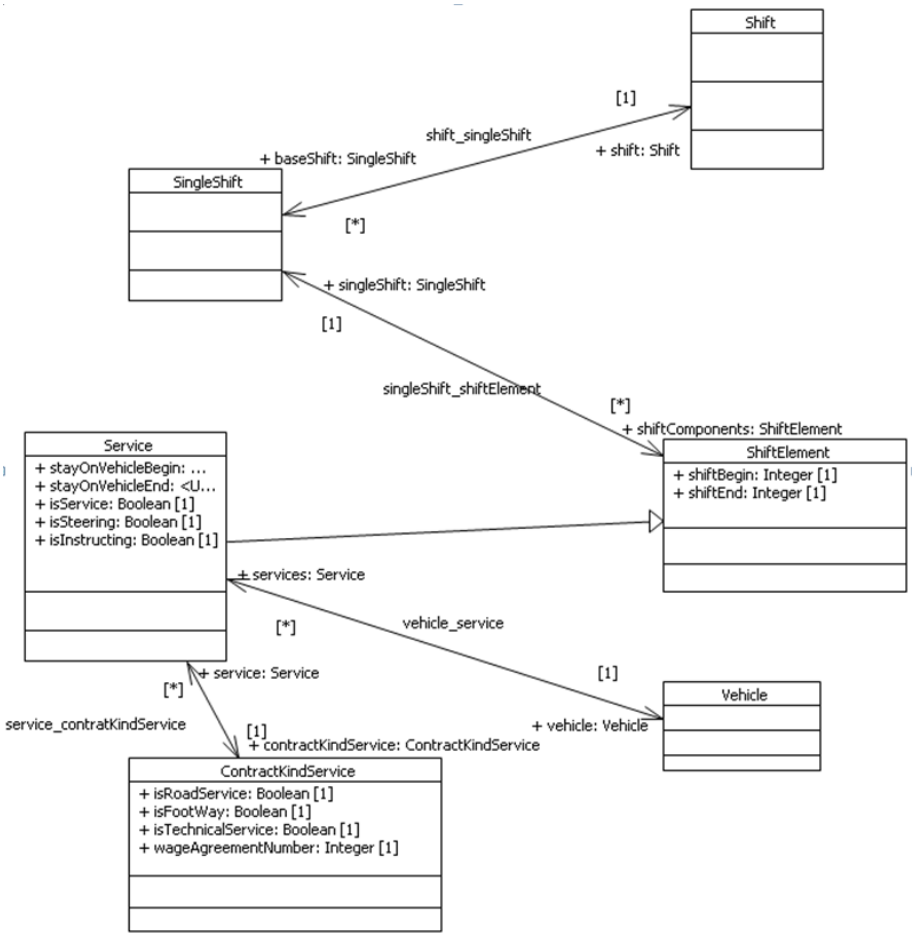


Fig. D.1: Partial Domain Model of Shift Planning

Appendix E

Supplementary Material on the Acceptance Study

For the acceptance study 30 students have been asked to reproduce formalised business rules in own words as accurate as possible. The content of the case study consists of a domain model (see Section E.1), which is the basis for the APRIL and OCL rules (see subset of both in sections E.2 and E.3). The APRIL and OCL rules can be sequentially matched one by one, whereas each APRIL-OCL-pair has the same semantics. The rules i1, i2 and i5 give an impression on what we consider to be simple business rules. Complex rules on the other hand are i13 - i16. Section E.4 gives an impression on the complexity of the task in which the students were supposed to formalize the respective business rule in APRIL and OCL.

E.1 Domain Model

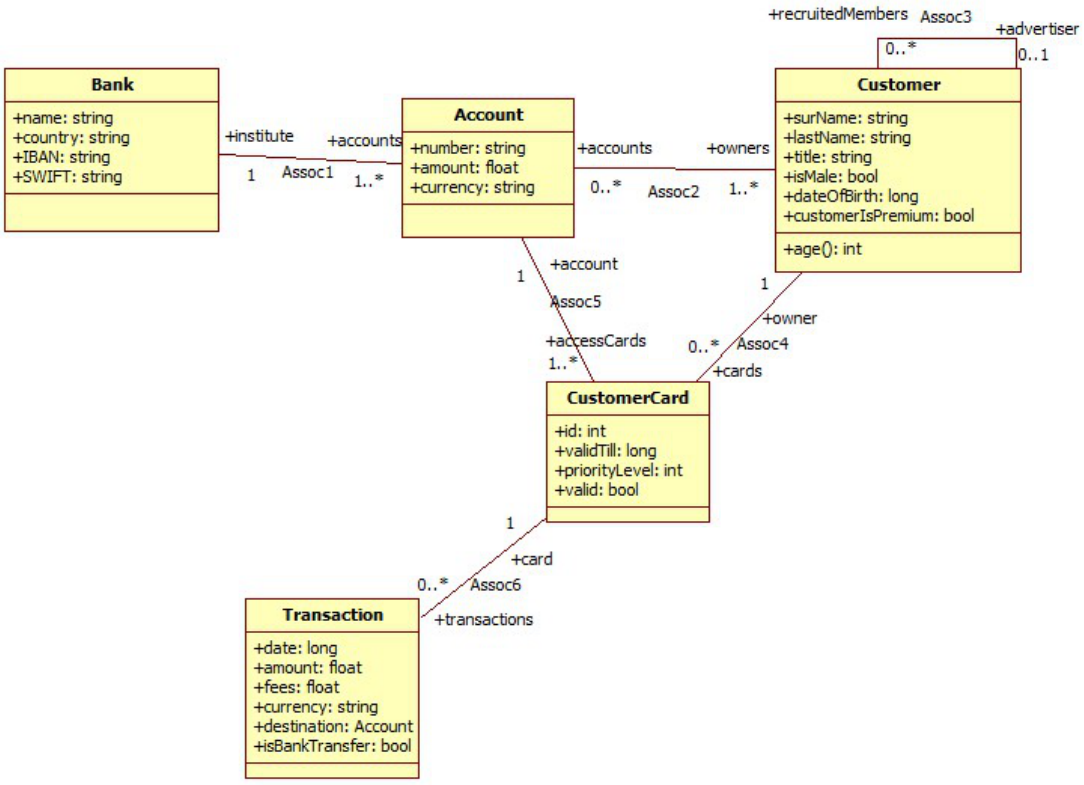


Fig. E.1: Domain model of the acceptance case study

E.2 APRIL Rules to Reproduce

```
Invariant i1 concerns Customer:
 age >= 18 .
```

```
Invariant i2 concerns Customer :
sum of cards.transactions.amount >= 10000 and sum of accounts.amount > 5000 .
```

```
Invariant i5 concerns Bank :
number is unique in accounts .
```

```
Invariant i13 concerns Customer:
this is not in reachable objects along recruitedMembers
```

```
Invariant i14 concerns Bank:
for every instance all or none of the following holds:
SWIFT is not empty , IBAN is not empty .
```

```
Invariant i15 concerns Account :
NoMoneySpendable implies that the owners have no cards
with
NoMoneySpendable is defined as
amount <= 0 .

Value the (customers as Collection of type Customer) have no cards
yielding Boolean is defined as
every customer in customers satisfies that number of cards = 0 .
```

```
Invariant i16 concerns Bank
every customer in WealthyMaleSeniorGermans satisfies that the customer donesnt have to pay fees for customer.
      cards.transactions to accounts of foreign banks
with
WealthyMaleSeniorGermans is defined as wealthy male senior accounts.owners in Germany .


```

```
Value the (customer as type Customer) donesnt have to pay fees for (transactions as Collection of type
      Transaction) yielding Boolean
is defined as
every transaction in CustomersTransactions satisfies that fees = 0
with
CustomersTransactions is defined as
each transaction in transactions where card.owner = customer .

Filter (transactions as Collection of type Transaction) to accounts of foreign banks yielding Collection of type
      Transaction
is defined as
each transaction in transactions where destination.institute.country <> 'Germany' .

Filter (customers as Collection of type Customer) in Germany yielding Collection of type Customer
is defined as
each customer in customers where at least one account in accounts satisfies that institute.country = 'Germany' .

Filter wealthy male senior (customers as Collection of type Customer) yielding Collection of type Customer is
      defined as
each customer in customers where age >=55 and isMale and sum of accounts.amount >= 1000000 .
```

E.3 OCL Rules to Reproduce

```
context Customer inv i1: age() >= 18
```

```
context Customer inv i2 :
cards.transactions.amount->sum() >= 10000 and
accounts.amount ->sum() > 5000
```

```
context Bank inv i5:
accounts->isUnique(number)
```

```
context Customer def :
successors(): Set(Customer) =self.recruitedMembers->union(self. recruitedMembers.successors())

context Customer inv i13:
self.successors()->excludes(self)
```

```
context Bank inv i14:
```

```
2 (self.IBAN->notEmpty() and self.SWIFT->notEmpty()) or not (self.IBAN->notEmpty() and self.SWIFT->notEmpty())
```

```
1 context Account inv i15:
2 self.amount <= 0 implies owners->forAll(customer | customer.cards->size()=0)
```

```
1 context Bank inv i16:
2 let wealthy_male_senior_customers : Collection(Customer) =  accounts.owners->select(customer | customer.age >= 55
        and customer.isMale and customer.accounts.amount->sum() >= 1000000) in
3 let WealthyMaleSeniorGermans : Collection(Customer) = wealthy_male_senior_customers->select(customer | customer.
      accounts->exists(account | account.institute.country='Germany') )
4 in
5 WealthyMaleSeniorGermans->forAll( customer | customer.cards.transactions->select(transaction | transaction.
      destination.institute.country <> 'Germany')->select(transaction | transaction.card.owner = customer)->forAll
      (transaction | transaction.fees=0))
```

E.4 Rules to Formalize

With respect to the domain model in Section E.1 the business rule to formalize were as follows:

- An object of the type *Bank* is identified by the attributes *name* and *country*.
- A customer cannot own more than three customer cards.
- A customer cannot have more than 2 customer cards at the same bank.

Appendix F

Use Case "Train Trip" of the ERTMS System

The following business rules are taken from the "Procedure Train Trip" sub chapter 5.11 of the mandatory system requirements specification for railway systems [29] of the European Union Agency For Railways. Any of the rules below is based on the related UML-class model in Figure F.1.

F.1 Business Rules and Domain Model

```
1 Precondition S010 concerns EEOnBoardEquipment.StartTrip() is defined as
2 mode.FS or mode.LS or mode.OS or mode.SR or mode.SB or mode.SH or mode.SN or mode.UN .
3
4 Postcondition S010 concerns EEOnBoardEquipment.StartTrip() is defined as
5 mode.TR = true.
```

Listing F.1.1: ERTMS / ECTS Train System, Use Case "Train Trip" Rule 1

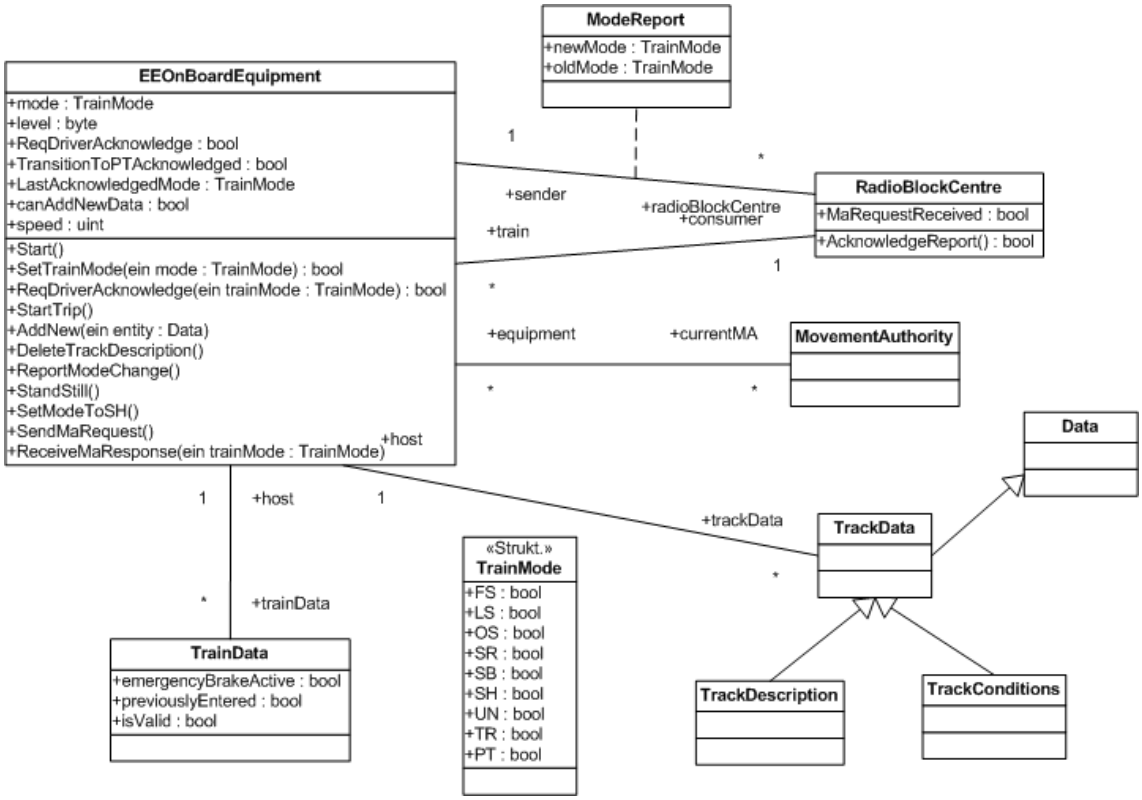


Fig. F.1: Domain Model of the ERTMS / ECTS Train System for the Driving Use Case

ID #	Requirements
S010	The ERTMS/ETCS on-board equipment is in one of the following modes: FS, LS, OS, SR, SB, SH, SN or UN When an event occurs, which leads to train trip reaction (**E015** – refer to chapter 4, transitions between modes), the process shall go to **A025**.
A025	The mode shall change to TR. The process shall go to **D020**.
D020	If the level is 1, the process shall go to **A035**. If the level is 2/3, the process shall go to **A030**. If the level is 0/NTC, the process shall go to **S050**.
A030	The ERTMS/ETCS on-board equipment shall report the mode change to the RBC The process shall go to **A035**.
A035	All current MA and track description data (if any), except track conditions, shall be deleted and new ones shall not be accepted The process shall go to **S050**.
S050	The ERTMS/ETCS on-board equipment awaits standstill. While braking a border to a level 0 or NTC area may be passed. When the train has come to standstill (**E055**), the process shall go to **S060**.
S060	The ERTMS/ETCS on-board equipment shall display the "Request for driver acknowledgement to Train Trip" to the driver. When the driver acknowledges the Train Trip (**E065**), the process shall go to **D080**.

Fig. F.2: Business rules of ERTMS Train Trip Specification from Rule S010 to Rule S060

```
Precondition A035 concerns EEOnBoardEquipment.DeleteTrackDescription() is defined as
level = 1.

Postcondition A035 concerns EEOnBoardEquipment.DeleteTrackDescription() is defined as
all currentMa including trackdata of type TrackDescription shall be deleted and canAddNewData = false.

Definition all (data as Collection of Data) shall be deleted yielding Boolean is defined as
data is empty.
```

Listing F.1.2: ERTMS / ECTS Train System, Use Case "Train Trip" Rule 2

```
Precondition A030 concerns EEOnBoardEquipment.ReportModeChange() is defined as
level = 2 or level = 3.

Postcondition A030 concerns EEOnBoardEquipment.ReportModeChange() is defined as
ModeReport.newMode = former value of mode.
```

Listing F.1.3: ERTMS / ECTS Train System, Use Case "Train Trip" Rule 3

```
Precondition S050 concerns EEOnBoardEquipment.StandStill() is defined as
level = 0.

Postcondition S060 concerns EEOnBoardEquipment.StandStill() is defined as
ReqDriverAcknowledge = true.

Postcondition S060 concerns EEOnBoardEquipment.ReqDriverAcknowledge(mode) is defined as
((level = 1 or level=2 or level=3) implies that (
mode.PT = true and revokeEmergencyBrake))
and
(level = 0 or levelIsNTC implies that
  if it is not the case that every data in trainData satisfies that data.isValid then mode.SH = true
 otherwise
  (level = 0 implies that mode.UN = true) and
  (level = 255 implies that mode.SN = true)
)
with
revokeEmergencyBrake is defined as each data in trainData satisfies that emergencyBrakeActive = false,
levelIsNTC is defined as level = 255 .
```

Listing F.1.4: ERTMS / ECTS Train System, Use Case "Train Trip" Rule 4

```
Definition pending emergency stops in (equipment as EEOnBoardEquipment) exist yielding boolean is defined as
each data in equipment.trainData satisfies that emergencyBrakeActive = true.

Precondition S140 concerns EEOnBoardEquipment.SetModeToSH() is defined as
it is not the case that pending emergency stops in this EEOnBoardEquipment exist.
```

```
6
7  Precondition S140 concerns EEOnBoardEquipment.Start() is defined as
8  level = 1 and trainData.previouslyEntered = true and it is not the case that pending emergency stops in this
         EEOnBoardEquipment exist.
9
10 Postcondition S160 concerns EEOnBoardEquipment.Start() is defined as
11 ReqDriverAchnowledge = true and (level = 2 or level = 3).
12
13 Precondition S160 concerns EEOnBoardEquipment.ReqDriverAcknowledge(trainMode) is defied as
14 trainMode.SR = true implies that ReqDriverAchnowledge = true.
15
16 Postcondition S160 concerns EEOnBoardEquipment.ReqDriverAcknowledge(trainMode) is defied as
17 trainMode.SR = true implies that mode.SR = true.
```

Listing F.1.5: ERTMS / ECTS Train System, Use Case "Train Trip" Rule 5

```
1  Precondition A115 concerns RadioBlockCentre.AcknowledgeReport() is defined as
2  train.mode = 2 or train.mode = 3.
3
4  Precondition A115 concerns RemoteBoardControl.AcknowledgeReport() is defined as
5  train.mode.TR = false.
```

Listing F.1.6: ERTMS / ECTS Train System, Use Case "Train Trip" Rule 6

```
1  Precondition S150 concerns EEOnBoardEquipment.SendMaRequest() is defined as
2  level = 2 or level = 3.
3
4  Postcondition S150 concerns EEOnBoardEquipment.SendMaRequest() is defined as
5  radioBlockCentre.MaRequestReceived = true and former value of RadioBlockCentre.MaRequestReceived = false.
6
7  Postcondition S150a concerns EEOnBoardEquipment.ReceiveMaResponse(trainMode as TrainMode) is defined as
8  trainMode.SR = true implies that mode.SR = true and ReqDriverAcknowledge = true.
9
10 Postcondition S150b concerns EEOnBoardEquipment.ReceiveMaResponse(trainMode as TrainMode) is defined as
11 (trainMode.OS or trainMode.LS or trainMode.SH) = true implies that ReqDriverAcknowledge = true.
12
13 Precondition S150b1 concerns EEOnBoardEquipment.ReqDriverAcknowledge(trainMode) is defied as
14 ReqDriverAcknowledge = true and (trainMode.OS implies that mode.OS and
15                                  trainMode.LS implies that mode.LS and
16                                  trainMode.SH implies that mode.SH).
17
18 Postcondition S150c concerns EEOnBoardEquipment.ReceiveMaResponse(trainMode as TrainMode) is defined as
19 trainMode.FS = true implies that mode.FS = true.
```

Listing F.1.7: ERTMS / ECTS Train System, Use Case "Train Trip" Rule 7

F.2 Categorisation of the Business Rules

This section contains the categorisation of the business rules in terms of complexity. The result is shown in Table F.1 that maps the line numbers in which the respective business rules to be categorised start to the listings they are contained in.

Number of Listing	Simple	Medium	Complex
F.1.1	5	2	
F.1.2	2,8	5	
F.1.3	2,5		
F.1.4	2,5		8
F.1.5	2,11,14,17	5,8	
F.1.6	2,5		
F.1.7	2,5,8,11,19		14

Table F.1: Categorisation of the Business Rules Complexity

Bibliography

[1] AARTS, F., AND AARTS, J. *English syntactic structures*. Pergamon Press, 1982.

[2] AHO, A., LAM, M., SETHI, R., AND ULLMAN, J. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, 2007.

[3] ANDROMDA.ORG. AndroMDA. `http://www.andromda.org/docs/index.html`, 2014. [Online; accessed 10-Feb.-2017].

[4] AWAD, A., WEIDLICH, M., AND WESKE, M. Visually specifying compliance rules and explaining their violations for business processes. *Journal of Visual Languages & Computing 22*, 1 (2011), 30–55.

[5] BACHERLER, C., FACCHI, C., AND WINDISCH, H.-M. Enhancing Domain Modeling with Easy to Understand Business Rules. `https://www.thi.de/fileadmin/daten/Working_Papers/thi_workingpaper_19_bacherler_facchi_windisch.pdf`, 2010. THI-Ingolstadt, Online; accessed: 10-Feb.-2017.

[6] BACHERLER, C., MOSZKOWSKI, B., AND FACCHI, C. Supporting Test Code Generation with an Easy to Understand Business Rule Language. *International Journal On Advances in Software 6*, 1 and 2 (2013), 69–79.

[7] BAJWA, I. S., BORDBAR, B., AND LEE, M. G. OCL Constraints Generation from Natural Language Specification: Enterprise Distributed Object Computing Conference (EDOC), 2010 14th IEEE International: Enterprise Distributed

Object Computing Conference (EDOC), 2010 14th IEEE International DOI - 10.1109/EDOC.2010.33. *Enterprise Distributed Object Computing Conference (EDOC), 2010 14th IEEE International* (2010), 204–213.

[8] BECK, K. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[9] BOLOTOV, A., BASUKOSKI, A., GRIGORIEV, O., AND SHANGIN, V. Natural deduction calculus for linear-time temporal logic. In *Logics in Artificial Intelligence*. Springer, 2006, pp. 56–68.

[10] BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. *Unified Modeling Language User Guide, The (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[11] BORLAND. Together. `https://www.borland.com/de-DE/Products/Requirements-Management/Together`, 2017. [Online; accessed 10-Feb.-2017].

[12] BUDINSKY, F., BRODSKY, S., AND MERKS, E. *Eclipse modeling framework*. Addison-Wesley Boston, 2003.

[13] BURKE, D. A., AND JOHANNISSON, K. Translating Formal Software Specifications to Natural Language. In *Logical Aspects of Computational Linguistics*, P. Blache, E. Stabler, J. Busquets, and R. Moot, Eds., vol. 3492 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 47–82.

[14] C. I. HOU, N. F. N. M. A. M. A Template-Based Approach Toward Acquisition of Logical Sentences: Intelligent Information Processing. *World Computer Congress* (2002).

[15] CABOT, J., PAU, R., AND RAVENTÓS, R. From UML/OCL to SBVR specifications: A challenging transformation. *Information Systems 35*, 4 (2010), 417–440.

[16] CABOT, J., AND TENIENTE, E. Transformation techniques for ocl constraints. *Science of Computer Programming 68*, 3 (2007), 179–195.

[17] CAMBRIA, E., AND WHITE, B. Jumping nlp curves: a review of natural language processing research [review article]. *IEEE Computational Intelligence Magazine 9*, 2 (2014), 48–57.

[18] CHOMSKY, N., AND SCHÜTZENBERGER, M. The Algebraic Theory of Context-Free Languages*. *Studies in Logic and the Foundations of Mathematics 35* (1963), 118–161.

[19] CHOW, T. S. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, 3 (1978), 178–187.

[20] COSTAL, D., GÓMEZ, C., QUERALT, A., RAVENTÓS, R., AND TENIENTE, E. Facilitating the Definition of General Constraints in UML. In *Model Driven Engineering Languages and Systems*, O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds., vol. 4199 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 260–274.

[21] CRISTIANINI, N., AND SHAWE-TAYLOR, J. *An introduction to support Vector Machines: and other kernel-based learning methods*. Cambridge university press, 2006.

[22] DANIELSSON, N., AND NORELL, U. Parsing mixfix operators. *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008)* (2009).

[23] DEMUTH, B., AND SNEED, H. Ein Modell für natursprachliche Anforderungsdokumente. *Informatik-Spektrum, Vol. 39* (2016), 362–372.

[24] DENGER, C., BERRY, D. M., AND KAMSTIES, E. Higher quality requirements specifications through natural language patterns. In *Proceedings. IEEE International Conference on : Software: Science, Technology and Engineering, 2003. SwSTE '03* (2003), pp. 80–90.

[25] DIJKSTRA, E. *A discipline of programming*, vol. 1. prentice-hall, 1976.

[26] ECLIPSE FOUNDATION, T. Eclipse modeling framework (emf). `https://eclipse.org/modeling/emf/`, 2015. [Online; accessed 12-02-2017].

[27] ECLIPSE OPEN PLATTFORM COMMUNITY. Xpand: Model To Text. `http://www.eclipse.org/modeling/m2t/?project=xpand`, 2010. [Online; accessed 12-Feb.-2017].

[28] ECLIPSE OPEN PLATTFORM COMMUNITY. AspectJ: Version 1.7.0. `http://www.eclipse.org/aspectj/`, 2012. [Online; accessed 12-Feb.-2017].

[29] EUROPEAN UNION AGENCY FOR RAILWAYS. Commission regulation (eu) 2016/919. `http://www.era.europa.eu/Document-Register/Documents/SUBSET-026%20v340.zip`, 2016. [Online; Accessed 09-03-2017].

[30] FERNÁNDEZ, D. M., AND WAGNER, S. Naming the pain in requirements engineering: A design for a global family of surveys and first results from Germany. *Information and Software Technology 57* (2015), 616–643.

[31] FISHER, M. *An Introduction to Practical Formal Methods Using Temporal Logic*. John Wiley & Sons, 2011.

[32] FOR RAILWAYS, E. U. A. System requirements specification. `http://www.era.europa.eu/Document-Register/Pages/`

Set-2-System-Requirements-Specification.aspx, 2007. [Online; Accessed 17-2-2017].

[33] FRANCE, R., AND RUMPE, B. Model-driven development of complex software: A research roadmap. In *FOSE '07 2007 Future of Software Engineering*. 2007.

[34] GE, R., AND MOONEY, R. A statistical semantic parser that integrates syntax and semantics. In *Ninth Conference on Computational Natural Language Learning*. 2005.

[35] GIORDANI, A., AND MOSCHITTI, A. Syntactic structural kernels for natural language interfaces to databases. *Machine Learning and Knowledge Discovery in Databases* (2009), 391–406.

[36] GOGOLLA, M., BÜTTNER, F., AND RICHTERS, M. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming 69*, 1 (2007), 27–34.

[37] GOGOLLA, M., AND RICHTERS, M. Expressing UML class diagrams properties with OCL. *Object Modeling with the OCL* (2002), 423–426.

[38] GOGOLLA, M., AND RICHTERS, M. Transformation rules for UML class diagrams. *The Unified Modeling Language. UML '98: Beyond the Notation* (2004), 514–514.

[39] GÓMEZ, R., AND BOWMAN, H. PITL2MONA: implementing a decision procedure for propositional interval temporal logic. *Journal of Applied Non-Classical Logics 14*, 1-2 (2004), 105–148.

[40] HALPIN, T. Business rule verbalization. *Information Systems Technology and its Applications, 3rd International Conference (ISTA'2004), LNI 48* (2004), 39–52.

[41] HALPIN, T. Objectification. In *Exploring Modeling Methods for Systems Analysis and Design* (2005), vol. 363, CEUR Workshop Proceedings, RWTH AACHEN University, pp. 133–146.

[42] HALPIN, T., MORGAN, A., AND MORGAN, T. *Information modeling and relational databases*. Morgan Kaufmann, 2008.

[43] HALPIN, T. A. Verbalizing Business Rules : Part 1-16. *Business Rules Journal* (2006).

[44] HART, G., JOHNSON, M., AND DOLBEAR, C. Rabbit: Developing a Control Natural Language for Authoring Ontologies. In *The Semantic Web: Research and Applications*, S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, Eds., vol. 5021 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 348–360.

[45] HINA AFREEN, I. S. B. Generating UML class models from SBVR software requirements specifications. *23rd Benelux Conference on Artificial Intelligence (BNAIC 2011)* (2011).

[46] HOUDEK, F., AND PAECH, B. *Das Türsteuergerät–eine Beispielspezifikation [The Car Door Control System–An Example Specification]*. Technical report, Fraunhofer Institut für Experimentelles Software Engineering, Kaiserslautern, 01.01.2002.

[47] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO Common Logic: ISO/IEC IS 24707:2007, 2007.

[48] ITEMIS. xtext. `http://www.itemis.de/itemis-ag/portfolio/eclipse-modeling/language=de/27262/xtext`, 2011. [Online; accessed 12-02-2017].

[49] JETBRAINS. mps: meta programming system. http://www.jetbrains.com/mps/, 2010. [Online; accessed 13-02-2017].

[50] JURAFSKY, D., MARTIN, J., NORVIG, P., AND RUSSELL, S. *Speech and Language Processing*. Pearson Education, 2014.

[51] KANSO, B., AND TAHA, S. Temporal constraint support for OCL. In *Software Language Engineering*. Springer, 2013, pp. 83–103.

[52] KATE, R., AND MOONEY, R. Using string-kernels for learning semantic parsers: Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics. 2006.

[53] KAZHAMIAKIN, R., PISTORE, M., AND ROVERI, M. Formal verification of requirements using SPIN: a case study on Web services. In *Proceedings of the Second International Conference on : Software Engineering and Formal Methods, 2004. SEFM 2004.* (2004), pp. 406–415.

[54] KEY PROJECT. The Key approach. https://www.key-project.org/, 2017. [Online; Accessed 17-2-2017].

[55] KLARLUND, N., AND MØLLER, A. *MONA Version 1.4 User Manual*. January 2001. Notes Series NS-01-1. Available from http://www.brics.dk/mona/. Revision of BRICS NS-98-3.

[56] KLEINER, M., ALBERT, P., AND BÉZIVIN, J. Parsing SBVR-Based Controlled Languages. In *Model Driven Engineering Languages and Systems*, A. Schürr and B. Selic, Eds., vol. 5795 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 122–136.

[57] KURTH, F., SCHUPP, S., AND WEISSLEDER, S. Generating test data from a uml activity using the ampl interface for constraint solvers. In *International Conference on Tests and Proofs* (2014), Springer, pp. 169–186.

[58] La Rosa, M., van der Aalst, W. M. P., Dumas, M., and Milani, F. P. Business process variability modeling: A survey. http://bpmcenter.org/wp-content/uploads/reports/2013/BPM-13-16.pdf, 2013. [Online; accessed 13-02-2017].

[59] Leveson, N. Completeness in formal specification language design for process-control systems. In *Proceedings of the third workshop on Formal methods in software practice* (2000), ACM, pp. 75–87.

[60] Linehan, M. Semantics in model-driven business design. In *2nd Semantic Web Policy Workshop SWPW 06* (2006), vol. 207, CEUR Workshop Proceedings, RWTH AACHEN University.

[61] Linehan, M. H. SBVR Use Cases. In *Rule Representation, Interchange and Reasoning on the Web*, N. Bassiliades, G. Governatori, and A. Paschke, Eds., vol. 5321 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, pp. 182–196.

[62] Liskov, B. Keynote address-data abstraction and hierarchy. *ACM Sigplan Notices 23*, 5 (1988), 17–34.

[63] Loucopoulos, P., and Karakostas, V. *System requirements engineering*. McGraw-Hill, Inc, 1995.

[64] Martin, J. *Rapid application development*. Macmillan Publishing Co., Inc, 1991.

[65] Matthiessen, G., and Unterstein, M. *Relationale Datenbanken und Standard-SQL: Konzepte der Entwicklung und Anwendung*. Pearson Education, 2007.

[66] Melenovsky, M. J. Business process management's success hinges on business-led initiatives. *Gartner Research, Stamford, CT* (2005), 1–6.

[67] MELLOR, S., AND BALCER, M. *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc, 2002.

[68] MENDLING, J., REIJERS, H. A., AND VAN DER AALST, W. M. P. Seven process modeling guidelines (7PMG). *Information and Software Technology 52*, 2 (2010), 127–136.

[69] METACASE. MetaEdit+. `http://www.metacase.com/de/mep/`, 2017. [Online; accessed 13-02-2017].

[70] MICH, L., FRANCH, M., AND NOVI INVERARDI, P. Market research for requirements analysis using linguistic tools. *Requirements Engineering 9*, 2 (2004), 151.

[71] MILIAUSKAIT, E., AND NEMURAIT, L. Representation of integrity constraints in conceptual models. *Information technology and control, Kauno technologijos universitetas, ISSN* (2005), 34ff.

[72] MILIAUSKAITE, E., AND NEMURAITE, L. Taxonomy of integrity constraints in conceptual models. *Proceedings of IADIS Database Systems* (2005).

[73] MISHRA, P., AND EICH, M. H. Join processing in relational databases. *ACM Computing Surveys (CSUR) 24*, 1 (1992), 63–113.

[74] MOSZKOWSKI, B. Executing temporal logic programs. In *Seminar on Concurrency*, S. Brookes, A. Roscoe, and G. Winskel, Eds., vol. 197 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1985, pp. 111–130.

[75] MOSZKOWSKI, B. A hierarchical completeness proof for propositional temporal logic. In *Verification: Theory and Practice*. Springer, 2003, pp. 480–523.

[76] MOSZKOWSKI, B. Interconnections between classes of sequentially compositional temporal formulas. *Information Processing Letters 113*, 9 (2013), 350–353.

[77] MOSZKOWSKI, B. Compositional reasoning using intervals and time reversal. *Annals of Mathematics and Artificial Intelligence 71*, 1-3 (2014), 175–250.

[78] MOSZKOWSKI, B. C. A complete axiomatization of interval temporal logic with infinite time. In *Logic in Computer Science, 2000. Proceedings. 15th Annual IEEE Symposium on : Logic in Computer Science, 2000.* (2000), pp. 241–252.

[79] NAVATHE, S., AND RAMEZ, E. *Fundamentals of Database Systems*. Addison-Wesley, 2002.

[80] NEBUT, C., FLEUREY, F., LE TRAON, Y., AND JÉZÉQUEL, J. Automatic test generation: A use case driven approach. *Software Engineering, IEEE Transactions on 32*, 3 (2006), 140–155.

[81] NELKEN, R., AND FRANCEZ, N. Automatic translation of natural language system specifications into temporal logic. In *Computer Aided Verification*, R. Alur and T. Henzinger, Eds., vol. 1102 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1996, pp. 360–371.

[82] NIJSSEN, M., AND LEMMENS, I. Verbalization for Business Rules and Two Flavors of Verbalization for Fact Examples. In *On the Move to Meaningful Internet Systems: OTM 2008 Workshops*, R. Meersman, Z. Tari, and P. Herrero, Eds., vol. 5333 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, pp. 760–769.

[83] OBJECT MANAGEMENT GROUP. SBVR Specification: version 1.0. `http://www.omg.org/spec/SBVR/1.0/`, 2008. [Online; Accessed 17-02-2017].

[84] OBJECT MANAGEMENT GROUP. UML Specification: Version 2.4.1. `http://www.omg.org/spec/UML/2.4.1/`, 2011. [Online; Accessed 17-02-2017].

[85] OBJECT MANAGEMENT GROUP. Object Constraint Langauge (OCL): Version 2.3.1. http://www.omg.org/spec/OCL/2.3.1/, 2012. [Online; Accessed 17-02-2017].

[86] OBJECT MANAGEMENT GROUP. Model Driven Architecture MDA. http://www.omg.org/mda/, 2017. [Online; Accessed 17-02-2017].

[87] OMG. Meta Object Facility: Specification. http://www.omg.org/spec/MOF/. [Online; Accessed 17-02-2017].

[88] OMG. UML Profile Specifications. http://www.omg.org/technology/documents/profile_catalog.htm. [Online; Accessed 17-02-2017].

[89] OMG. Query View Transformation: QVT. http://www.omg.org/spec/QVT/1.0/, 2008. [Online; Accessed 17-02-2017].

[90] OSTRAND, T. J., AND BALCER, M. J. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM 31*, 6 (1988), 676–686.

[91] PARR, T. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007.

[92] PARR, T. ANTLR v3. http://www.antlr.org/, 2012. [Online; accessed: 06-06-2012].

[93] PARR, T. String Template: Version 4.0. http://www.stringtemplate.org/, 2012. [Online, Accessed: 06-06-2012].

[94] PLAISTED, D. A., AND GREENBAUM, S. A structure-preserving clause form translation. *Journal of Symbolic Computation 2*, 3 (1986), 293–304.

[95] PNUELI, A. The temporal logic of programs. *Foundations of Computer Science, 1977., 18th Annual Symposium on* (1977).

[96] PNUELI, A. In Transition From Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems*, K. Apt, Ed., vol. 13 of *NATO ASI Series*. Springer Berlin Heidelberg, 1985, pp. 123–144.

[97] QUINLAN, J. R. *C4. 5: programs for machine learning*. Elsevier, 2014.

[98] RUPP, C. *Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis*, 5., aktualisierte und erw ed. Hanser, München and Wien, 2009.

[99] RUPP, C. *Requirements-Engineering und -Management: Aus der Praxis von klassich bis agil*, 6., aktualisierte und erw ed. Hanser, München, 2014.

[100] RUPP, C., QUEINS, S., AND ZENGLER, B. *UML 2 glasklar: Praxiswissen für die UML-Modellierung*, 3., aktualisierte aufl. ed. Hanser, München and and Wien, 2007.

[101] SAFONOV, V. *Using aspect-oriented programming for trustworthy software development*. Wiley Online Library, 2008.

[102] SALEMI, S., SELAMAT, A., AND PENHAKER, M. A model transformation framework to increase {OCL} usability. *Journal of King Saud University - Computer and Information Sciences 28*, 1 (2016), 13 – 26.

[103] SCHACHER, M., AND GRÄSSLE, P. *Agile Unternehmen durch Business Rules. Der Business Rules Ansatz*. Springer, Berlin Heidelberg New York, 2006.

[104] SENDALL, S., AND STROHMEIER, A. Using OCL and UML to specify system behavior. In *Object Modeling with the OCL*. Springer, 2002, pp. 250–279.

[105] SJIR NIJSSEN. SBVR: Semantics for Business. *Business Rules Journal, Vol 8. No 10* (2007). URL: `http://www.brcommunity.com/p-b367.php`.

[106] SOSUNOVAS, S., AND VASILECAS, O. Precise notation for business rules templates. *Databases and Information Systems, 2006 7th International Baltic Conference on* (2006), 55–60.

[107] SPREEUWENBERG, S., AND HEALY, K. SBVR's approach to controlled natural language. *Controlled Natural Language* (2010), 155–169.

[108] SPREEUWENBERG, S., VAN GRONDELLE, J., HELLER, R., AND GRIJZEN, G. Using CNL techniques and pattern sentences to involve domain experts in modeling. *Controlled Natural Language* (2012), 175–193.

[109] STAHL, T., AND VÖLTER, M. Model-driven software development. *Technology, Engineering, Management. England: John Wiley & Sons* (2006).

[110] STANDISH GROUP. THE STANDISH GROUP REPORT: CHAOS. `http://www.projectsmart.co.uk/docs/chaos-report.pdf`, 2014. [Online; Accessed 17-02-2017].

[111] STEEDMAN, M. *The syntactic process*, vol. 24. MIT Press, 2000.

[112] TETKO, I. V., LIVINGSTONE, D. J., AND LUIK, A. I. Neural network studies. 1. Comparison of overfitting and overtraining. *Journal of Chemical Information and Computer Sciences 35*, 5 (1995), 826–833.

[113] TJONG, S. F., HALLAM, N., AND HARTLEY, M. Improving the Quality of Natural Language Requirements Specifications through Natural Language Requirements Patterns. *CIT 6* (2006), 199–205.

[114] TU BREMEN. USE: Uml-based specification environment. `https://sourceforge.net/projects/useocl/`. [Online; Accessed 17-2-2017].

[115] TU DRESDEN. Dresden OCL Toolkit. `http://www.dresden-ocl.org/index.php/DresdenOCL`, 2011. [Online; Accessed 17-2-2017].

[116] UTTING, M., PRETSCHNER, A., AND LEGEARD, B. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab. 22*, 5 (Aug. 2012), 297–312.

[117] VAN DER AALST, W. Geschäftsprozessmodellierung: Die Killer-Applikation für Petrinetze. *Informatik-Spektrum 37*, 3 (2014), 191–198.

[118] VAN GRONDELLE, J., HELLER, R., VAN HAANDEL, E., AND VERBURG, T. Involving business users in formal modeling using natural language pattern sentences. *Knowledge Engineering and Management by the Masses* (2010), 31–43.

[119] VAN LAMSWEERDE, A. *Requirements engineering: from system goals to UML models to software specifications*. Wiley, Chichester, 2009.

[120] VARDI, M. Y. Branching vs. linear time: Final showdown. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2001, pp. 1–22.

[121] WARMER, J., AND KLEPPE, A. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Longman Publishing Co., Inc, 2003.

[122] WOLFGANG THOMAS. A combinatorial approach to the theory of *omega*-automata. *Information and Control 48*, 3 (1981), 261–283.

[123] WONG, Y., AND MOONEY, R. Learning for semantic parsing with statistical machine translation: Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics. In *Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*. 2006.

[124] WOODCOCK, J., LARSEN, P., BICARREGUI, J., AND FITZGERALD, J. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR) 41*, 4 (2009), 1–36.

[125] XU, D. A tool for automated test code generation from high-level petri nets. In *International Conference on Application and Theory of Petri Nets and Concurrency* (2011), Springer, pp. 308–317.

[126] ZETTLEMOYER, L., AND COLLINS, M. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proc. of UAI*, vol. 5. 2012.